UNIVERSITÉ DE MONTRÉAL

POINTS DE TRACE STATIQUES ET DYNAMIQUES EN MODE NOYAU

RAFIK FAHEM

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAITRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AVRIL 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

POINTS DE TRACE STATIQUES ET DYNAMIQUES EN MODE NOYAU

présenté par : FAHEM, Rafik

en vue de l'obtention du diplôme de : Maitrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme. BOUCHENEB Hanifa, Doctorat, présidente

M. DAGENAIS Michel, Ph.D., membre et directeur de recherche

M. MERLO Ettore, Ph.D., membre

REMERCIEMENTS

Je tiens à témoigner toute ma reconnaissance à mon directeur de recherche, Monsieur Michel Dagenais qui m'a honoré en m'acceptant au sein de l'équipe DORSAL et en me confiant ce projet. Sa disponibilité, son écoute attentive et l'aide financière qu'il m'a versée m'ont été d'une aide précieuse.

Je tiens aussi à remercier Mathieu Desnoyers pour avoir partagé son expérience très riche dans le domaine du traçage et pour ses conseils qui ont été utiles.

Enfin, je souhaiterai rendre hommage aux membres de ma famille qui m'ont soutenu moralement tout au long de mes études.

RÉSUMÉ

En utilisant TRACE_EVENT et UST, il est maintenant possible d'insérer des points d'instrumentation statiques sous Linux en modes noyau et usager respectivement, tout en minimisant l'impact sur la performance du système instrumenté. Toutefois, ces points d'instrumentation peuvent parfois s'avérer insuffisants pour diagnostiquer les origines d'un problème. L'instrumentation dynamique répond à ce besoin en rendant possible l'insertion de points de trace supplémentaires au moment de l'exécution.

Récemment, les points de trace dynamiques ont été implémentés dans GDB et GDBServer en mode usager. En utilisant cette technique, on est capable d'associer un ensemble d'actions à n'importe quelle adresse dans le code du programme. Ces actions peuvent servir à collecter les valeurs des registres au moment où le point de trace est rencontré et aussi à évaluer des expressions complexes qui peuvent employer les variables accessibles à partir de cet endroit dans le programme. GDB étant capable de lire les informations de débogage et de localiser l'emplacement de chaque variable, on peut faire référence à une variable dans ces expressions en utilisant directement son nom sans se soucier de son emplacement. Les points de trace statiques et dynamiques de GDB peuvent être conditionnels. Dans ce cas, une expression est utilisée comme condition. Les expressions utilisées dans les conditions et expressions sont converties en code intermédiaire qui est interprété par GDBServer au moment où le point de trace est rencontré. L'intérêt du code intermédiaire est de garantir la portabilité entre les différentes architectures. Cependant, GDBServer peut le transformer en code natif afin d'améliorer la performance. En effet, exécuter du code natif est souvent plus rapide que d'avoir à identifier et exécuter des instructions une par une.

Plus récemment, le module KGTP a été proposé comme contribution au noyau Linux. Il se base sur Kprobes pour implémenter les points de trace dynamiques de GDB en mode noyau et communique avec celui-ci en utilisant le protocole RSP (Remote Serial Protocol). Il est seulement capable d'interpréter le code intermédiaire produit par GDB et ne peut pas faire de conversion en code natif.

L'objectif de ce travail est d'implémenter un convertisseur en mode noyau pour KGTP pour traduire le code intermédiaire en code natif pour les conditions et aussi les actions afin

d’améliorer la performance des points de trace dynamiques. Aussi, nous allons intégrer TRACE_EVENT et GDB en mode noyau à travers KGTP pour être capable, tout comme en mode usager, de lister, activer et désactiver les points de trace statiques du noyau. Le même convertisseur de code intermédiaire à code natif est utilisé avec les points de trace statiques pour pouvoir leur associer des conditions et des expressions supplémentaires à exécuter. Ces expressions doivent elles aussi être capables d’utiliser toutes les variables accessibles au niveau du point de trace statique.

ABSTRACT

With kernel static tracepoints defined using TRACE_EVENT and user-space tracepoints through the UST library, it is now possible to add instrumentation and obtain a low overhead trace of the whole system. However, these static tracepoints may be insufficient to diagnose the source of a problem. Dynamic instrumentation fills the gap by making it possible to insert additional tracepoints in other locations at run time.

Recently, GDB was enhanced to support dynamic tracepoints in user-space. Using this feature, tracepoints can be defined in almost every location in a program. A set of actions can be associated to each tracepoint. These actions may be used to collect the values of the registers at the time the tracepoint was hit or to evaluate user-defined expressions. These expressions may be complex and can employ all the program variables accessible from the tracepoint location. GDB being able to read the program debug information and to locate variables, we can refer to variables in these expressions by their names without having to care about their locations. GDB static and dynamic tracepoints may be conditional. In this case, expressions can be used as conditions. In order to simplify evaluation, GDB converts expressions used in conditions and actions to bytecode which is interpreted each time the corresponding tracepoint is hit. Moreover, in some situations, GDB converts the conditions' bytecodes into native code in order to improve performance.

More recently, the KGTP kernel module was submitted as a contribution to the Linux kernel. It uses Kprobes to insert GDB dynamic tracepoints into the kernel, implements the RSP (Remote Serial Protocol) to communicate with GDB and can interpret the bytecode used by GDB to define conditions and actions, but is unable to convert this bytecode to native code.

The goal of this work is to extend the KGTP module by implementing a bytecode to native code converter in kernel space for both conditions and actions. GDB will also be integrated with TRACE_EVENT through KGTP in order to be able to list, enable and disable the kernel static tracepoints. Expressions may be used in conditions and additional actions. These expressions will be converted to native code and may employ all the variables accessible from the static tracepoint location.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

GDB	GNU Debugger
RSP	Remote Serial Protocol
KGTP	Kernel GDB Tracepoints
LTTng	Linux Trace Toolkit Next Generation

INTRODUCTION

Problème étudié et buts poursuivis

Le traçage[1] est une technique permettant l'enregistrement d'événements se produisant au sein d'un système dans le but d'identifier les causes de problèmes de fonctionnement ou de performance[2, 3]. Il présente une alternative efficace aux techniques de débogage et profilage classiques dans le cas où le système à analyser possède une architecture complexe ou possède des contraintes temporelles dures de telle sorte qu'on ne peut pas l'interrompre pour observer son fonctionnement sans risquer de changer son comportement.

Le traçage consiste à ajouter du code d'instrumentation au programme. Ce code permet de collecter certaines données qui seront utiles pour analyser le système. Il existe deux types de points de trace : dynamiques et statiques.

Les points de trace dynamiques sont définis et insérés au moment de l'exécution du programme. Le code d'instrumentation est inséré en effectuant des modifications au code binaire. Le traçage dynamique a l'avantage de ne causer de temps d'exécution supplémentaire que lorsque les points de trace sont activés. Par contre, ce type de traçage nécessite une connaissance approfondie du code du programme instrumenté afin de bien choisir l'emplacement du point de trace et de déterminer les données qu'on peut extraire.

Les points de trace statiques sont insérés au niveau du code du programme avant la compilation. Ils sont toujours définis dans le programme et peuvent être activés et désactivés au besoin. Les points de trace statiques permettent d'analyser un système sans avoir forcément de connaissance approfondie du code. Par contre, ce type de traçage entraine un temps d'exécution additionnel par rapport au code original même quand le point de trace est désactivé. En effet, il y a toujours un code additionnel exécuté et qui vérifie si le point de trace est activé ou pas.

Un point de trace, qu'il soit dynamique ou statique, peut être conditionnel, c'est-à-dire qu'une fois rencontré, une condition est évaluée. Les données ne sont collectées que si la condition est vraie.

Sous Linux, et en mode usager, il existe plusieurs outils qui offrent les deux types de traçage. Nous pouvons citer par exemple GDB. Cet outil de débogage permet d'insérer des points de trace dynamiques et de se connecter aux points de trace statiques définis dans le code à l'aide de la librairie UST. Les points de trace dynamiques peuvent être insérés pratiquement à n'importe quelle adresse du programme. Ils peuvent servir à collecter les registres du processeur et les variables du programme accessibles à partir de l'adresse du point de trace. Surtout, ils sont capables d'évaluer des expressions complexes pouvant employer les variables du programme. Les points de trace statiques peuvent être listés, activés et désactivés. En plus des données statiques à collecter définies à l'aide de la librairie UST[4], ils offrent toutes les autres fonctionnalités des points de trace dynamiques, à savoir l'évaluation d'expressions dynamiques et la collecte de registres. Les deux types de points de trace peuvent être conditionnels et, dans ce cas, les conditions sont définies à l'aide d'expressions. GDB étant capable de lire les informations de débogage, les expressions à collecter et les expressions utilisées dans les conditions peuvent faire référence aux variables directement avec leur nom.

Pour accélérer l'évaluation de ces expressions, GDB les traduit sous forme de code intermédiaire qui est interprété à chaque fois que le point de trace est rencontré. De plus, ce code intermédiaire est transformé en code natif si certaines conditions sont vérifiées afin d'accélérer encore plus l'évaluation.

En mode noyau[5], il existe des outils de traçage qui implémentent les points de trace dynamiques et qui sont capables de se connecter aux points de trace statiques définis à l'aide de TRACE_EVENT. Ces outils sont soit peu flexibles soit possèdent des fonctionnalités limitées par rapport à l'implémentation des points de trace dans GDB en mode usager. À titre d'exemple, Ftrace n'est pas capable de lire les informations de débogage générées lors de la compilation du noyau et n'est donc pas capable de localiser les variables. C'est pour cette raison que l'on doit spécifier manuellement l'emplacement de chaque variable qu'on veut collecter à partir d'un point de trace dynamique. Aussi, les deux types de points de trace ne peuvent être conditionnels.

SystemTap est un autre outil de traçage sur Linux. Il permet la définition de points de trace dynamiques et la collecte de données à partir des points de trace statiques définis à l'aide de TRACE_EVENT. Le code d'instrumentation est écrit sous forme de scripts. Quoiqu'il soit capable de lire les informations de débogage, il nécessite l'utilisation d'un compilateur pour

insérer le code d'instrumentation sous forme de module noyau. En plus, à chaque changement du code d'instrumentation, il faut refaire le processus de compilation. Finalement, SystemTap n'est pas capable de connaitre les données à extraire à partir des points de trace statiques automatiquement en se basant sur la définition du TRACE_EVENT. Il faut donc redéfinir ces données manuellement dans le script.

Récemment, le projet KGTP[6] (Kernel GDB Tracepoints) a été proposé comme contribution au noyau Linux. Il implémente les points de trace dynamiques de GDB en mode noyau. Il remplace donc GDBServer en mode noyau. Il s'agit d'un module qui, une fois inséré dans le noyau, est capable de communiquer avec GDB en utilisant le protocole RSP (Remote Serial Protocol). Ce protocole permet d'implémenter des éléments de remplacement de GDB pour des systèmes et des architectures non supportées par GDB. KGTP n'est capable que d'interpréter le code intermédiaire des expressions et ne peut pas le transformer en code natif comme c'est le cas avec GDBServer en mode usager.

Le but de ce projet est de développer un outil qui implémente les points de trace dynamiques dans le noyau Linux et qui est capable de se connecter aux points de trace statiques définis avec TRACE_EVENT. Les deux types de points de trace doivent être conditionnels. Les conditions seront définies sous forme d'expressions employant les variables du code. De même, on doit être capable d'évaluer et collecter des expressions à partir des deux types de points de trace. Les expressions à collecter et celles utilisées pour les conditions peuvent être complexes et doivent avoir une syntaxe proche de celle du langage C.

Une grande importance doit être accordée à la performance de l'outil. En effet, les délais apportés par le traçage ne doivent avoir qu'un impact mineur sur le système à tracer.

Démarche de l'ensemble du travail

L'approche envisagée est de se baser sur le module KGTP et de le compléter pour implémenter les deux types de traçage. Vu que toutes les fonctionnalités recherchées pour les points de trace dynamiques (évaluation d'expressions, collecte de registres) sont déjà existantes, nous allons nous concentrer sur l'aspect de la performance. Pour ce faire, nous allons implémenter un

convertisseur qui va transformer le code intermédiaire produit par GDB pour les expressions, en code natif. L'architecture cible est x86_64.

Ensuite, nous allons intégrer KGTP avec les points de trace définis dans le noyau avec TRACE_EVENT. Pour ce faire, nous allons devoir déterminer la manière pour les lister, et les activer et désactiver. Aussi, nous allons déterminer la méthode pour collecter les données statiques telles que définies dans les TRACE_EVENT. Ensuite, nous allons implémenter la collecte des registres au niveau du point de trace. Ces registres seront utilisés pour évaluer les expressions à évaluer et celles utilisées dans les conditions. Finalement, nous allons étudier l'efficacité des structures de données utilisées par KGTP pour sauvegarder la trace et nous allons proposer une méthode pour accélérer le traçage dans le cas où on découvre que ces structures entrainent des délais considérables.

Organisation générale du document

Au premier chapitre, nous allons présenter une revue de littérature dans laquelle nous discutons des techniques existantes pour les points de trace dynamiques et statiques. Nous allons nous concentrer sur les techniques que nous allons utiliser pour implémenter l'outil, et sur les autres outils de traçage qui offrent des fonctionnalités proches de celles qu'on veut avoir, à savoir la collecte de registres, l'évaluation d'expressions dynamiques et la collecte de données statiques. Le chapitre 2 est sous forme d'article dans lequel nous allons présenter la méthodologie suivie ainsi que les résultats que nous avons obtenus. Le chapitre 3 présente les résultats supplémentaires. Finalement, le chapitre 4 consiste en une discussion complémentaire et propose des travaux futurs.

CHAPITRE 1 REVUE DE LITTÉRATURE

Dans ce chapitre, nous allons présenter l'état de l'art des techniques existantes de traçage statique et dynamique. Ces techniques comprennent les mécanismes que nous allons utiliser dans le noyau Linux pour insérer des points de trace dynamiques et statiques. Aussi, nous allons présenter une revue des principaux outils de traçage qui sont utilisés dans Linux en modes usager et noyau ainsi que d'autres systèmes tels que Solaris. Ensuite, nous allons introduire des techniques utilisées pour la manipulation dynamique de programmes pour ajouter du code d'instrumentation. Finalement, nous allons présenter les objectifs de ce travail.

1.1 Kprobes

Kprobes[7, 8] est un mécanisme de débogage développé par IBM. Il faisait partie d'un outil de traçage de plus haut niveau appelé Dprobes. Kprobes a été intégré dans le noyau Linux depuis la version 2.6.9. Il s'agit d'une interface permettant d'insérer dynamiquement des points d'instrumentation dans le noyau au moment de l'exécution.

Le principe de kprobes est simple[9]. Il s'agit d'associer un événement à une adresse dans le code du noyau. À chaque fois que l'exécution arrive à cette adresse, l'événement est lancé. Kprobes est capable de s'insérer dans la quasi-totalité du code du noyau et permet d'associer une fonction à exécuter à chaque événement. Il existe trois types de kprobes : les kprobes ordinaires, les kretprobes et les jprobes.

Les kprobes ordinaires peuvent être insérés dans n'importe quelle routine du noyau. Pour chaque événement, on pourrait associer deux fonctions. La première, appelée pre_handler, est exécutée avant l'instruction se situant à l'adresse du point d'instrumentation. La deuxième, appelée post_handler est exécutée après. Quand le pre_handler et le post_handler sont appelés, une copie de l'état de tous les registres au moment où l'événement s'est produit leur est passée dans une structure de type pt_regs. Cette structure change de contenu d'une architecture à l'autre et contient entre autres les registres à usage général du processeur.

Les kretprobes sont enregistrés aux points de retour des fonctions. Les jprobes, quant à eux, peuvent être associés aux points d'entrée des fonctions du noyau et servent à collecter les arguments de ces fonctions. Dans ce qui suit, nous allons nous concentrer sur les kprobes

ordinaires vu qu'ils sont les plus utilisés dans les outils de traçage. Cette discussion portera seulement sur l'architecture 80x86, où les instructions sont de tailles variables.

Quand un kprobe est enregistré, l'instruction se situant au point d'instrumentation est copiée et remplacée par une instruction int3. Quand cette instruction est rencontrée, une interruption est lancée et une copie des registres est effectuée. À ce moment, la fonction pre_handler est appelée. Au retour de cette fonction, l'instruction initiale se trouvant à cette adresse est exécutée. Finalement, kprobes lance une deuxième interruption(single-step trap)[10] afin d'appeler la fonction post_handler. Le principe de kprobes est illustré dans la figure ci-dessous.

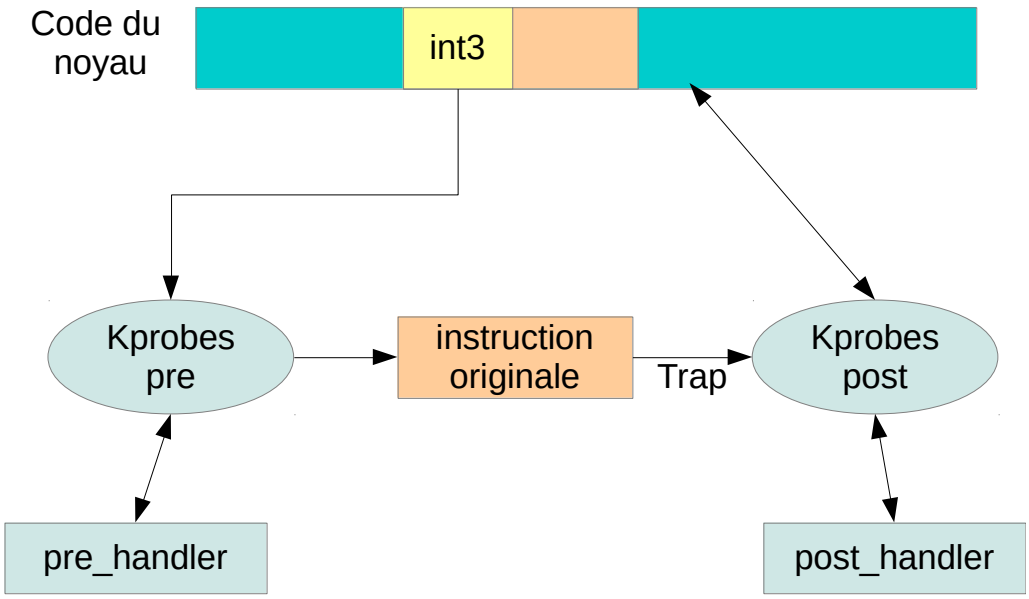

Figure 1- kprobes non optimisés

À cause des deux interruptions lancées, cette implémentation des kprobes souffre d'un problème de performance. C'est pour cela qu'il a fallu chercher des optimisations si on voulait minimiser l'impact sur le système qu'on est en train d'instrumenter. Les « fast kprobes » présentent une solution à ce problème. Au lieu d'insérer l'instruction int3 au niveau du point d'instrumentation, une instruction « jump » est utilisée[11]. Quand le point d'instrumentation est rencontré, au lieu de générer une interruption, l'instruction permet de sauter au code qui effectue la collecte des registres suivi du pre_handler. Au retour de cette fonction, l'instruction écrasée est exécutée. Finalement, on retourne à l'instruction qui suit le point d'instrumentation sans appeler de post_handler. La figure ci-dessous illustre ce mécanisme.

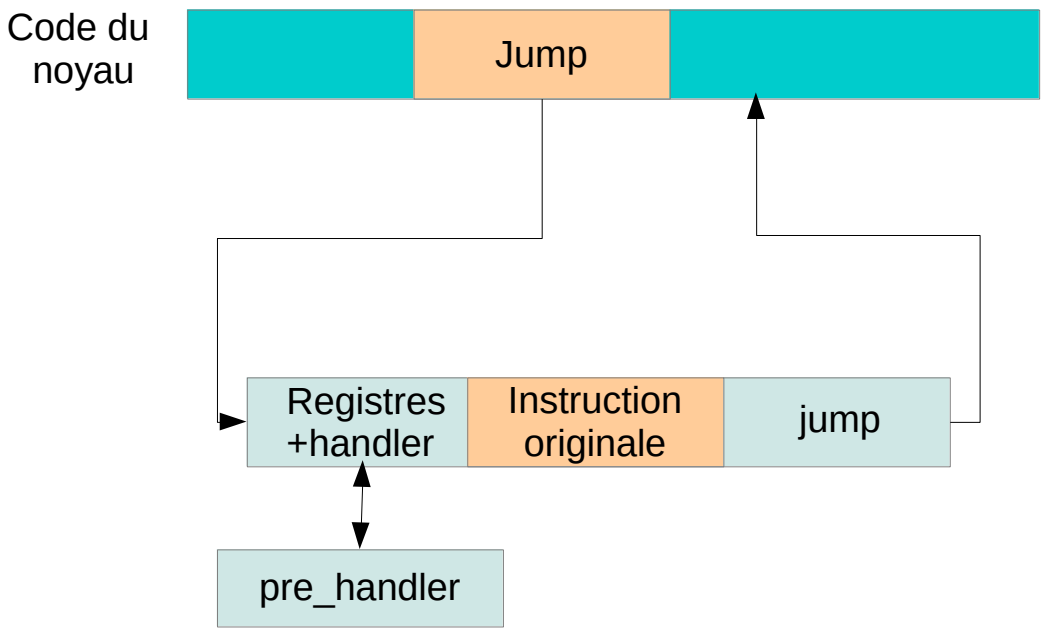

Figure 2 - fast kprobes

Il n'est pas toujours possible d'effectuer cette optimisation. En effet, dans le cas où l'instruction à écraser a une longueur inférieure à 5 octets, il n'est pas possible d'insérer l'instruction jump à la place de l'ancienne instruction (car une instruction jump a une longueur de 5 octets). En plus, on ne doit pas avoir spécifié de post_handler pour le kprobe. Il faut noter que la limite de 5 octets a été diminuée à 3 octets dans les dernières versions de kprobes en utilisant des « near jump ». La figure suivante illustre ce mécanisme.

Dans le cas où cette optimisation n'est pas réalisable, kprobes tente d'effectuer une autre optimisation. Il s'agit des « boosted kprobes ». Ce type de kprobes est inséré à l'aide d'une instruction int3 mais la deuxième interruption pour appeler le post_handler est omise. Il faut noter que cette optimisation n'est pas toujours réalisable elle aussi, comme dans le cas où l'instruction originale est un « call » ou un « near jump » par exemple.

Ces deux niveaux d'optimisation permettent d'obtenir des performances nettement supérieures à celle des kprobes non optimisés.

Les fonctions register_kprobe et unregister_kprobe servent à activer et désactiver un kprobe. Ces fonctions prennent en paramètre une variable de type « struct kprobe ». Ce paramètre permet de spécifier l'adresse du point d'instrumentation à insérer, les adresses des fonctions à appeler au

niveau du pre_handler et du post_handler. La structure possède d'autres champs comme celui de l'adresse de la fonction à appeler en cas d'erreur. En général, les appels à ces fonctions se font à l'intérieur de modules noyau.

Les kprobes sont utilisés dans la plupart des outils de traçage noyau récents tels que Ftrace, LTTng et SystemTap.

1.2 TRACE_EVENT

Plusieurs efforts ont été faits pour essayer de trouver une façon pour permettre aux développeurs de définir des points de trace statiques dans le noyau[12-14]. Contrairement aux points de trace dynamiques, ceux-ci sont définis au niveau du code et non pas en effectuant des modifications binaires au moment de l'exécution. Le code d'instrumentation est ainsi compilé avec le code du noyau.

TRACE_EVENT est le mécanisme le plus récent qui implémente les points de trace statiques dans le noyau. Il a été proposé comme solution de rechange aux « trace markers » et aux « tracepoints » qui étaient jugés pas assez faciles à intégrer avec les outils de traçage. De plus, ils ressemblaient à des printk et mélangeaient le code du noyau avec le code d'instrumentation. Au lieu de définir les points de trace et les données qu'ils doivent collecter au niveau du code, TRACE_EVENT permet de déplacer ces définitions dans des fichiers d'en-tête. Le code ajouté au niveau d'un point d'instrumentation est constitué d'un seul appel à une fonction, à laquelle on passe les arguments, tel que défini dans le TRACE_EVENT. Cette fonction va à son tour appeler toutes les fonctions enregistrées à ce point de trace. Il s'agit des fonctions enregistrées par les outils de traçage. Ainsi, on a minimisé les changements à effectuer au niveau du code, tout en augmentant la flexibilité et la facilité d'intégration avec les outils de traçage.

La définition[15] d'un point de trace statique se fait en appelant la macro TRACE_EVENT(). Cette macro prend 6 paramètres :

- name : correspond au nom du point de trace.
- proto : le prototype des fonctions qu'on pourra connecter au point de trace.
- args : les arguments qui correspondent au prototype.
- struct : la structure qui servira à extraire les données à collecter à partir des arguments

passés au callback.

- assign : les assignations nécessaires pour remplir la structure.
- print : la façon dont les données extraites dans la structure seront affichées.

Chaque paramètre est défini à son tour à l'aide d'une macro. Ceci nous permet de fournir plus qu'un élément par paramètre et aussi de rendre ce mécanisme assez flexible pour qu'il soit utilisé par les traceurs.

La macro TRACE_EVENT est définie initialement dans le fichier include/linux/tracepoint.h. Cette macro permet de définir, à partir de l'appel de la macro, les fonctions permettant aux outils de traçage de s'inscrire et se désinscrire d'un point de trace ainsi que la fonction à appeler au niveau du point d'instrumentation. Le paramètre "name" est utilisé pour définir les noms de ces fonctions pour les différencier les unes des autres. Ainsi, chaque point de trace possède sa propre fonction d'enregistrement. Les paramètres "proto" et "args" servent dans les définitions des prototypes et les corps de ces fonctions.

Les autres paramètres sont ignorés à cette étape. Ils peuvent être utilisés plutôt par les outils de traçage pour définir la manière dont les données sont extraites à partir des arguments et représentées à l'usager. Ftrace et Perf utilisent ces macros par exemple.

La figure 3 présente un exemple de TRACE_EVENT utilisé pour déclarer un point de trace statique utilisé dans la fonction load_module dans le fichier kernel/module.c. Le point de trace est déclaré dans include/trace/events/module.h.

```
TRACE_EVENT(module_load,

    TP_PROTO(struct module *mod),

    TP_ARGS(mod),

    TP_STRUCT__entry(
        __field(    unsigned int,   taints      )
        __string(   name,       mod->name   )
    ),

    TP_fast_assign(
        __entry->taints = mod->taints;
        __assign_str(name, mod->name);
    ),

    TP_printk("%s %s", __get_str(name),
show_module_flags(__entry->taints))
);
```

Figure 3 - Exemple de TRACE_EVENT

Le point de trace déclaré s'appelle module_load. Les callbacks qui peuvent s'y enregistrer doivent avoir le prototype (struct module *mod). L'argument passé à ces calbacks doit s'appeler mod.

À partir de cet appel, TRACE_EVENT va définir trois fonctions:

- register_trace_module_load(void (*probe)(data_proto), void *data): permet aux outils de traçage de s'inscrire au point de trace.
- unregister_trace_module_load(void (*probe)(data_proto), void *data): permet aux outils de traçage de se désinscrire.
- trace_module_load(struct module *mod): cette fonction représente la seule modification au niveau du code du noyau. Elle doit être placée du niveau du point d'instrumentation et permet d'appeler les fonctions inscrites par les traceurs.

Les macros TP_STRUCT__entry, TP_fast_assign et TP_printk sont ignorées à cette étape. En effet, TRACE_EVENT laisse aux développeurs d'outils de traçage le soin de redéfinir ces macros pour faciliter l'intégration. À titre d'exemple, Ftrace redéfinit ces macros quatre fois. Après chaque redéfinition, le fichier d'en-tête est ré-inclus pour produire le code souhaité. Ces quatre étapes servent à définir la fonction de Ftrace qui se connecte à ce point de trace ainsi que toutes les structures intermédiaires nécessaires pour extraire les données à partir des paramètres de la fonction.

1.3 GDB et GDBServer

GDB (GNU Debugger) est le débogueur standard du projet GNU. Il permet d'examiner un programme en cours d'exécution ou d'identifier la cause d'un crash. GDB offre une fonctionnalité très intéressante qui est les "tracepoints". Il s'agit d'un mécanisme permettant d'analyser un programme sans l'interrompre. En effet, dans certains cas, le programme à déboguer possède des contraintes temporelles très fortes de telle sorte que si on veut l'interrompre pour laisser à l'usager le temps pour l'examiner, on risque de changer son comportement.

En utilisant les commandes "trace" et "collect"[16], on peut spécifier des endroits dans le programme et des actions à exécuter lorsque ces endroits sont atteints. GDB offre la possibilité d'associer des conditions à ces actions. Il existe trois types d'actions: collecte de registres, collecte

de variables et évaluation d'expressions. Dans ce qui suit, nous allons nous intéresser plus à l'évaluation d'expressions.

Les expressions à évaluer et les expressions utilisées pour les conditions sont écrites dans une syntaxe très proche de celle du langage C. Elles peuvent employer les variables du programme accessibles à partir de l'endroit auquel on associe le point de trace. Elles peuvent aussi utiliser la plupart des opérations logiques et arithmétiques ainsi que les opérateurs de manipulation de bits.

Actuellement, les tracepoints GDB ne sont implémentés qu'en mode distant. Si on veut les utiliser en mode usager, il faut absolument utiliser GDBServer même si on veut déboguer en local. Cette fonctionnalité est disponible depuis la version 7.2 de GDB/GDBServer. GDB, lui, ne s'occupe que de la définition du tracepoint et des actions. Il utilise les données de débogage du programme pour vérifier leur validité. Ce n'est qu'au début de la session d'instrumentation que ces définitions sont envoyées à GDBServer.

GDB et GDBServer communiquent ensemble en utilisant un protocole propre à GDB qui s'appelle RSP (Remote Serial Protocol)[17]. Ce protocole ne se limite pas aux tracepoints mais il est utilisé pour toute communication entre les deux outils.

Les tracepoints GDB utilisent un déroutement (trap) pour insérer le point de trace dans le programme. Quand celui-ci est rencontré, une exception est lancée. L'état des registres est alors sauvegardé et le handler de GDBServer est appelé pour exécuter les actions. Le principe est très proche de celui de Kprobes.

Tout comme Kprobes, les tracepoints GDB peuvent être optimisés en utilisant une instruction jump au lieu de l'exception. Il s'agit des “fast tracepoints”[18]. Quand un tracepoint rapide est rencontré, le saut nous amène au handler de GDBServer qui exécute les actions. Les fast tracepoints sont définis à l'aide de la commande “ftrace”. Les actions sont définies à l'aide des mêmes commandes utilisées pour les tracepoints ordinaires.

Pour que l'instruction jump puisse appeler le handler de GDBServer, il faut que cette fonction soit dans l'espace d'adressage du programme à déboguer. C'est pour cette raison qu'on utilise un agent appelé “In-Process Agent” inséré dans le programme. Cet agent est inséré sous forme de librairie dynamique au moment de la compilation du programme. Il s'occupe entre autres de l'exécution

des tracepoints rapides et effectue le traçage dans des tampons propres à lui. GDBServer communique avec l'agent par scrutation pour lire la trace.

Pour éviter d'implémenter tout un interpréteur d'expressions au niveau de la machine cible(GDBServer) pour évaluer les conditions des tracepoints et les expressions définies dans les actions, et dans le souci de faciliter la portabilité, GDB utilise un langage intermédiaire pour représenter ces expressions[19]. Ainsi, avant de commencer l'instrumentation, GDB traduit les expressions à collecter et les expressions utilisées pour définir les conditions associées aux tracepoints en code intermédiaire qui sera interprété par GDBServer. La transmission de ce code intermédiaire se fait également en utilisant le protocole RSP, avant de commencer le traçage.

Le langage intermédiaire est composé d'une quarantaine d'opérations. Chaque opération est représentée par un code d'une longueur d’un octet qui peut être suivi de un ou plusieurs octets pour les arguments. GDBServer reçoit le code intermédiaire des expressions à collecter et celui des conditions et le copie tel quel. À chaque fois que le programme atteint un endroit où un point de trace est défini, GDB interprète le code de la condition du tracepoint si elle existe. Si l'expession évaluée est vraie ou s'il n'y a pas de condition, les actions associées au tracepoint sont exécutées. Sinon, si la condition existe et se révèle fausse, GDBServer ignore le tracepoint et retourne à l'exécution du programme. Une pile est utilisée pour sauvegarder les résultats temporaires des opérations du code intermédiaire ainsi que le résultat final.

Dans le cas où l'on a défini des expressions à évaluer dans la liste d'actions, GDBServer les interprète tout comme la condition. Il faut noter que GDBServer ne sauvegarde pas la valeur obtenue après l'évaluation de l'expression. Il existe plutôt des opérations dans le langage intermédiaire qui permettent de copier la mémoire dans les tampons utilisés pour le traçage. Il s'agit des opérations “trace” et “trace_quick”. Ces instructions sont utilisées pour sauvegarder la valeur de chaque variable utilisée dans l'expression à part. Ces valeurs sauvegardées sont utilisées plus tard par GDB pour réévaluer l'expression afin d'afficher la trace à l'usager.

Pour illustrer cela, nous avons tracé un programme très simple qui incrémente deux variables “i” et “j” et qui affiche les nouvelles valeurs par la suite. Après avoir démarré GDBServer en spécifiant le port 1234 comme port de communication, nous avons lancé GDB et établi la

communication sur ce port. Les commandes utilisées pour définir le point de trace sont les suivantes:

- trace main.c:9 if (i+j >=0)
- actions
    - collect (2*i+9*j)
    - end

Nous avons ainsi défini un tracepoint au niveau de la ligne 9 du fichier main.c en lui associant la condition (i+j>=0). Après, nous avons défini l'action qui consiste à évaluer et collecter l'expression (2*i+9*j).

Il faut noter que GDB ne définit pas la manière dont la trace est sauvegardée et que dans notre cas, celle-ci est sauvegardée dans les tampons de GDBServer. À chaque fois que l'usager demande à lire la trace, GDB envoie une requête à GDBServer en utilisant le protocole RSP.

Le tableau suivant présente le code intermédiaire[20] correspondant à la condition utilisée. Les instructions et leurs arguments sont représentés en hexadécimal.

Tableau 1 - code intermédiaire de la condition

Code	**Instruction**	**Description**
0x26 0x0 0x6	reg	Mettre la valeur du registre numéro 6 sur la pile. Il s'agit du registre RBP.
0x22 0x10	const8	Mettre la constante 0x10 qui est d'une longueur de un octet sur la pile
0x2	add	Enlever les deux entiers du sommet de la pile, les additionner et mettre le résultat sur la pile.
0x22 0xe8	const8	Mettre la constante 0xe8 sur la pile
0x16 0x8	ext	Enlever l'entier se trouvant au sommet de la pile tout en le considérant comme étant une valeur de 8 bits en complément à deux et l'étendre à la longueur d'un entier, c'est-à-dire mettre tous les bits se trouvant à gauche du bit 7(le bit 0 étant le moins significatif) à la valeur du bit 7. La nouvelle valeur est mise sur la pile.
0x2	add	-
0x19	ref32	Enlever l'adresse se trouvant au sommet de la pile et mettre à sa place

		l'entier de 32 bits se trouvant à cette adresse. Cet entier correspond à la valeur de la variable i.
0x16 0x20		Mettre la constante 0x20 sur la pile.
0x26 0x0 0x6	reg	-
0x22 0x10	const8	-
0x2	add	-
0x22 0xec	const8	-
0x16 0x8	ext	-
0x2	add	-
0x19	ref32	Cette fois-ci, l'entier se trouvant au sommet de la pile correspond à la variable j.
0x16 0x20	ext	Enlever l'entier se trouvant au sommet de la pile tout en le considérant comme étant une valeur de 32 bits en complément à deux pour l'architecture x86(ou 64 bits pour l'architecture x86_64) et l'étendre à la longueur d'un entier, c'est-à-dire mettre tous les bits se trouvant à gauche du bit 31(le bit 0 étant le moins significatif) à la valeur du bit 31. La nouvelle valeur est mise sur la pile.
0x2	add	-
0x16 0x20	ext	-
0x22 0x0	const8	-
0x14	less_signed	Enlever les deux entiers se trouvant au sommet de la pile. Si le premier entier retiré est plus grand que le deuxième, mettre la valeur 1 sur la pile. Sinon, mettre la valeur 0.
0xe	log_not	Enlever l'entier se trouvant au sommet de la pile, lui appliquer un NON logique et mettre le résultat sur la pile.
0x27	end	Arrêter l'interprétation du code intermédiaire. Le résultat de l'expression évaluée devrait se trouver au sommet de la pile.

De même, nous présentons dans le tableau suivant le code intermédiaire correspondant à l'expression à collecter (2*i+9*j). Comme pour le premier tableau, les instructions et leurs arguments sont représentés en hexadécimal.

Tableau 2 - code intermédiaire d'une expression à collecter

Code	**Instruction**	**Description**
0x22 0x2	const8	Il s'agit de la constante multiplicative 2.
0x26 0x0 0x6	reg	
0x22 0x10	const8	
0x2	add	
0x22 0xe8	const8	
0x16 0x8	ext	
0x2	add	
0xd 0x4	trace_quick	Copier 4 octets à partir de l'adresse correspondant à l'entier se trouvant au sommet de la pile. L'adresse est conservée sur la pile. Les données sont copiées dans les tampons de GDBServer. Dans notre cas, les 4 octets copiés correspondent à la valeur de la variable i.
0x19	ref32	La variable i est chargée dans la pile.
0x16 0x20	ext	
0x4	mul	Enlever les deux entier se trouvant au sommet de la pile, les multiplier l'un par l'autre et mettre le résultat sur la pile.
0x16 0x20	ext	
0x22 0x9	const8	Constante multiplicative 9.
0x26 0x0 0x6	reg	
0x22 0x10	const8	
0x2	add	
0x22 0xec	const8	
0x16 0x8	ext	
0x2	add	
0xd	trace_quick	Très similaire à la première instruction trace_quick, sauf que dans ce cas-ci, il s'agit de la variable j.
0x4	mul	
0x19	ref32	
0x16 0x20	ext	
0x4	mul	
0x16 0x20	ext	

0x2	add	
0x16 0x20	ext	
0x29	pop	Enlever l'entier se trouvant au sommet de la pile.
0x27	end	

Comme nous pouvons le constater, le code intermédiaire produit par GDB est plus facile à interpréter que les expressions initiales. Ceci permet d'alléger l'outil de débogage qui roule sur la cible (GDBServer dans notre cas). En outre, le code intermédiaire est indépendant de l'architecture de la machine cible. Ainsi, pour ajouter le support pour une certaine architecture, il suffit d'implémenter un interpréteur du langage intermédiaire de GDB.

Le langage intermédiaire de GDB présente donc une solution portable et légère pour l'évaluation d'expressions. Néanmoins, il possède les défauts de tout langage interprété. Il faut s'attendre donc à ce que le code intermédiaire soit plus lent à exécuter que du code natif.

Pour remédier à ce problème, GDBServer convertit les expressions utilisées pour les conditions des tracepoints en code natif. La raison pour laquelle cette opération est effectuée seulement pour les conditions et non pas pour les actions est, selon les développeurs, que les délais entrainés par le traçage des variables dans les tampons est tellement significatif qu'il importe peu de les convertir. Ainsi, au lieu d'avoir à interpréter le code intermédiaire à chaque fois que le tracepoint est rencontré, le code natif est exécuté directement. Cette optimisation est implémentée pour les tracepoints rapides seulement. Aussi, au lieu d'utiliser une pile comme dans le cas du code intermédiaire et qui serait difficile à manipuler à partir du code assembleur, la pile du programme elle-même est utilisée. Ceci ne présente aucun risque sur l'exactitude du programme vu que le code natif produit est sous forme de fonction et que la pile revient à son état initial à la sortie de la fonction.

L'idée de la traduction en code natif est très simple: pour chaque instruction du code intermédiaire, il existe un bout de code natif écrit en assembleur qui lui correspond. Avant de commencer le traçage, GDBServer alloue un tampon dans zone mémoire exécutable pour chaque expression. Ensuite, on parcourt le code intermédiaire et, pour chaque instruction, le code natif correspondant est copié dans le tampon. À la fin du parcours, le tampon est transformé en

fonction en appliquant un forçage de type. Ainsi, il suffit d'appeler cette fonction si on veut vérifier la condition au lieu d'interpréter le code intermédiaire.

Les tracepoints ordinaires et les tracepoints rapides sont des points de trace dynamiques. Ils sont définis au moment de l'exécution et peuvent être insérés à n'importe quel endroit du code du programme. Il existe une troisième catégorie de tracepoints dans GDB. Il s'agit des tracepoints statiques. Contrairement aux points de trace dynamiques, ceux-ci sont définis au niveau du code. Le code d'instrumentation est compilé avec celui du programme. Les points de trace statiques supportés par GDB sont ceux insérés à l'aide de la librairie UST (User-Space Tracer) qui fait partie du projet LTTng. Les points de trace statiques sont similaires aux TRACE_EVENT utilisés dans le noyau linux. Ils permettent de collecter les données avec un appel similaire à un printf. Les données collectées sont sauvegardées dont les tampons de LTTng. GDB est capable de se connecter à ces points de trace et, au lieu de faire la collecte dans LTTng, les données sont sauvegardées dans les tampons de l'agent In-Process Agent. En plus des données statiques, GDB est aussi capable, tout comme pour les autres types de tracepoints, de collecter les registres, les variables et les expressions. Aussi, des expressions peuvent être utilisées comme condition.

Vu que les tracepoints statiques sont définis au moment du codage, on n'a pas à les redéfinir avec GDB. GDB permet seulement de les lister, activer, désactiver et de leur associer les actions à exécuter.

1.4 KGTP

GDB n'implémente les tracepoints qu'en mode distant, et la communication avec l'élément de remplacement (stub) sur la machine distante se fait avec le protocole RSP. Ainsi, si on veut ajouter le support des tracepoints sur une certaine architecture ou pour un certain système, tel que les systèmes temps-réel, il suffit de développer un élément de remplacement qui supporte cette architecture ou ce système et qui est capable de communiquer correctement avec GDB en utilisant le protocole RSP. Ainsi, GDBServer est un élément de remplacement de GDB en mode usager.

À partir de cette idée, le projet KGTP[6] a vu le jour. Il s'agit d'un module noyau qui permet d'implémenter le support des tracepoints en mode noyau. La version que nous avons utilisée tout

au long de ce projet supporte seulement les tracepoints dynamiques. Ceux-ci sont définis à l'aide de la commande “trace”. Le code intermédiaire généré par GDB est interprété sans passer par la génération de code natif. Le module supporte les architectures x86_32, x86_64, ARM et MIPS. Les points de trace conditionnels sont aussi supportés par KGTP.

KGTP utilise Kprobes pour insérer les points de trace dynamiques. Dépendamment de l'instruction se trouvant à l'adresse du point de trace, Kprobes va appliquer les optimisations possibles. Cette optimisation est hors du contrôle de KGTP. Pour KGTP, il s'agit toujours d'un tracepoint ordinaire.

Le module enregistre une seule fonction comme handler de tous les kprobes insérés dans le noyau. Quand un point de trace dynamique est rencontré, Kprobes appelle cette fonction et lui passe le paramètre de type « struct kprobe » qui lui a été fourni au moment de l'enregistrement du point de trace, en plus de la copie des registres. À partir de cette variable, KGTP est capable d'identifier le point de trace qui a été rencontré. En effet, il garde une copie de cette variable dans une structure plus grande de type « gtp_entry ». KGTP utilise cette structure pour représenter un point de trace. Ainsi, en appelant la macro container_of, KGTP est capable d’accéder à la structure « gtp_entry » et d'identifier le point de trace dynamique en question. La structure trouvée contient une copie du code intermédiaire de la condition du point de trace ainsi que ceux de toutes les actions à exécuter.

KGTP enregistre la trace dans son propre tampon. La trace est transmise à GDB en utilisant le protocole RSP au moment de sa lecture. Pour tracer une expression, comme pour le mode usager, KGTP enregistre la valeur de chaque variable utilisée dans l'expression à part. GDB réévalue l'expression au moment de la lecture de la trace par après.

Le tampon de la trace est structuré en frames de tailles différentes. À chaque fois qu'une donnée doit être enregistrée dans la trace, qu'il s'agisse de la copie des registres ou la copie d'une variable utilisée dans une expression, KGTP crée un nouveau frame. D'abord, un entête est inséré au début du frame pour spécifier le type des données à copier ainsi que la taille de ces données. Cet entête est suivi de la donnée elle-même. En plus, KGTP vérifie, avant la création de chaque frame, qu'il reste encore de l'espace disponible dans les tampons. Le temps requis pour ces vérifications ainsi que pour effectuer les copies risque d’entraîner des délais considérables pendant le traçage.

1.5 DTrace

DTrace[21] est un traceur dynamique implémenté pour la première fois dans Solaris 10 et développé par Sun Microsystems. Depuis, DTrace a été porté aux plateformes Mac OS, QNX et FreeBSD. Un port pour la plateforme Oracle Enterprise Linux est en cours de développement. L'aspect dynamique dans DTrace vient du fait que les points d'instrumentation peuvent être activés seulement en cas de besoin et peuvent être désactivés lorsque les données demandées sont collectées[22].

La définition d'un point d'instrumentation est composée de quatre éléments:

- Le provider ou fournisseur: il s'agit d'un module noyau qui regroupe des points d'instrumentation logiquement inter-reliés. On peut citer par exemple le fournisseur FBT qui instrumente les fonctions du noyau.
- Le module: il s'agit de l'endroit où les points d'instrumentation se situent. Il pourrait correspondre par exemple au nom d'un module noyau.
- La fonction: la fonction à laquelle on veut associer le point d'instrumentation.
- Le nom: il s'agit de la signification du point d'instrumentation. Pour les fonctions, on pourrait utiliser “entry” ou “return” pour insérer le point de trace au début ou à la fin de la fonction instrumentée.

DTrace est formé de plusieurs composants répartis entre l'espace noyau et l'espace usager. Il permet entre autres l'instrumentation dynamique et statique du noyau. La collecte des informations se fait à l'aide d'un module noyau.

L'instrumentation dynamique permet d'éviter l'effet d'un point d'instrumentation statique qui induit un petit délai même lorsqu'il est désactivé. Ce mode est implémenté dans le fournisseur FBT (Function Boundary Provider) qui offre un point d'instrumentation au début et à la fin de la quasi-totalité des fonctions du noyau. Ainsi, le nombre de points disponibles dépasse 25,000. Il est à noter que le traçage dynamique se limite à ces points seulement et qu'il est impossible d'insérer des points d'instrumentation supplémentaires dans d'autres endroits, contrairement à d'autres outils de traçage tels que SystemTap.

Les points de trace dynamiques sont implémentés sur l'architecture x86 en utilisant un déroutement ou trap. Une fois rencontrée, cette instruction transfère le contrôle à la table de description d'interruptions(IDT) qui, à son tour, donne le contrôle à DTrace. Cette instruction remplace une des instructions qui crée le stack frame si le point de trace est inséré à l'entrée de la fonction(ou une des instructions qui le détruisent dans le cas où on est à la sortie de la fonction). Une fois qu'on retourne de DTrace, l'instruction écrasée est exécutée.

Les points de trace statiques sont implémentés dans le fournisseur SDT(Statically Defined Tracing). Un point d'instrumentation est inséré en faisant un appel à une macro C qui se traduit par un appel à une fonction non existante. Quand l'éditeur de liens découvre que la fonction est inexistante, il remplace l'appel par une opération NOP et enregistre le nom de la fonction appelée ainsi que l'adresse de l'appel. Pour activer un point de trace statique, DTrace consulte cette structure pour donner le nom approprié à la fonction et remplace l'instruction NOP par un appel à cette fonction.

Néanmoins, le compilateur doit insérer le code pour préparer l'appel à la fonction, ce qui implique l'utilisation de registres supplémentaires même si la fonction est inexistante. Cet effet est inévitable. Par contre, il a été limité en plaçant les points d'instrumentation tout près des appels à d'autres fonctions dans le code du noyau.

Le code d'instrumentation est écrit sous forme de scripts, en utilisant le langage de haut niveau D de DTrace. Il permet de définir les conditions et les actions à exécuter pour chaque point de trace activé. Les conditions sont formulées sous forme de prédicats qui sont vérifiés à chaque fois que le point de trace est rencontré. Si ces prédicats sont vrais, les actions sont exécutées. Celles-ci servent à spécifier les données à collecter.

La syntaxe du langage D est très proche de celle du langage C et de awk[23] sauf que les conditions sont définies à l'aide de prédicats et non pas en utilisant des structures conditionnelles dans les scripts. D supporte tous les opérateurs d'ANSI C[24]. Les scripts DTrace peuvent employer les variables accessibles à partir de l'adresse du point d'instrumentation. L'information sur les symboles est extraite en utilisant un service spécial dans le noyau.

Les scripts D sont convertis dans un format appelé DIF (D Intermediate Format) en utilisant un compilateur spécial en mode usager. Ce langage intermédiaire est constitué d'un petit jeu

d'instructions RISC conçu pour être facile à émuler. Une fois le code intermédiaire produit, il est envoyé à l'espace noyau pour que DTrace le vérifie et active le point de trace qui lui correspond.

1.6 SystemTap

SystemTap[25-27] est un outil de traçage similaire à DTrace[28] permettant la collecte d'information à partir d'un système Linux en cours d'exécution, dans le but de faciliter l'identification des causes de problèmes fonctionnels ou de performance. Cet outil permet l'insertion de points de trace dynamiques ainsi que la collecte de données à partir des points de trace statiques définis à l'aide de TRACE_EVENT. La trace collectée peut être affichée sur la console au fur et à mesure qu'elle est produite. Sinon, le mode flight recorder peut enregistrer la trace soit en mémoire soit dans un fichier temporaire pour qu'elle soit analysée plus tard.

Tout comme DTrace, le code d'instrumentation de SystemTap est écrit sous forme de scripts[29]. Le langage utilisé a une syntaxe très proche de celle de C. Il supporte toutes les opérations de ANSI C. Les types de données supportés sont seulement les entiers et les chaînes de caractères. Un script SystemTap peut servir à déclarer plusieurs points d'instrumentation.

La déclaration d'un point d'instrumentation est composée de deux parties[30]. La première partie sert à identifier l'événement auquel on veut associer le point d'instrumentation. La deuxième partie correspond au code à exécuter lorsque le point de trace est rencontré. Les points de trace peuvent être conditionnels. Pour définir une condition, on doit utiliser un « if » au niveau du script. La figure 4 montre un exemple de script SystemTap :

```
probe kernel.function("vfs_read")
{
    dev_nr = $file->f_path->dentry->d_inode->i_sb-
>s_dev
    if(dev_nr >= 3)
        printf ("%x\n", dev_nr)
}
```

Figure 4 - Exemple de script SystemTap

Le point d'instrumentation est inséré à l'entrée de la fonction vfs_read. L'information est extraite à partir du paramètre « file » de la fonction et est copiée dans une variable temporaire. Cette variable est ensuite utilisée dans la condition et dans l'expression à collecter.

Les scripts SystemTap sont convertis en code C, qui est compilé par après sous forme de module noyau. Ce module est alors inséré et communique avec SystemTap pour le traçage. Toutefois, il est possible de passer en mode « guru » où il est possible d'insérer du code C dans les scripts. Ceci permet de dépasser certaines limites du langage de script.

Les points de trace dynamiques de SystemTap sont implémentés en utilisant Kprobes. Ils peuvent être conditionnels. Les conditions sont spécifiées avec des « if » comme dans l'exemple précédent. SystemTap se base sur l'information de débogage DWARF, générée lors de la compilation du noyau, pour déterminer les adresses des points d'instrumentation ainsi que pour résoudre les références vers les variables du noyau utilisées dans les scripts. SystemTap utilise alors les registres passés au handler du kprobe pour extraire les valeurs de ces variables. Les points de trace dynamiques peuvent utiliser toutes les variables accessibles à partir de l'adresse du point d'instrumentation.

Les points de trace dynamiques peuvent être insérés dans la quasi-totalité du code du noyau. SystemTap offre une multitude d'événements auxquels on peut associer des points d'instrumentation. On peut en citer les suivants :

- Les points d'entrée et de sortie de toutes les fonctions traçables du noyau. Exemples :

probe kernel.function("nom_de_la_fonction").call

probe kernel.function("nom_de_la_fonction").return

La première déclaration sert à insérer le point de trace à l'entrée de la fonction alors que la seconde correspond à la sortie de la fonction.

- Un certain emplacement dans le code du noyau. Exemple :

probe kernel.statement("*@fs/read_write.c:309")

- Une certaine adresse dans le binaire. Exemple :

probe kernel.statement(0xc0044852)

SystemTap permet aussi de se connecter aux points de trace statiques définis avec TRACE_EVENT. L'événement est déclaré dans le script de la façon suivante : probe kernel.trace("nom_evenement"). Le nom de l'événement correspond au nom donné au point de trace statique lors de l'appel à la macro TRACE_EVENT. Les points de trace statiques peuvent eux aussi être conditionnels. Les conditions sont spécifiées en utilisant un « if » dans le script. Contrairement à celles des points de trace dynamiques, celles-ci ne peuvent employer que les variables passées en paramètre au niveau du point d'instrumentation. Ceci est dû au fait que les handlers des points de trace statiques ne reçoivent pas de copie des registres au moment où l'événement est lancé. Il est alors impossible d'aller chercher les valeurs des autres variables.

Ce problème est aussi rencontré dans les expressions à collecter. Celles-ci ont les mêmes limitations que les conditions. En plus, SystemTap ne peut pas utiliser le format d'affichage des données extraites à partir de ces paramètres, tel que défini dans le TRACE_EVENT à l'aide de la macro TP_printk. En effet, SystemTap ne manipule pas les macros de TRACE_EVENT comme décrit dans la section 1. Il ne se base que sur les informations de débogage pour localiser les points de trace statiques et les paramètres passés à chacun d'entre eux. Ainsi, SystemTap ne fait que donner accès au point de trace. Pour représenter les données telles que définies dans le TRACE_EVENT, il faut redéfinir la façon de le faire dans le script.

1.7 Ftrace

Ftrace[31] est un ensemble de traceurs faisant partie du noyau Linux depuis sa version 2.6.27. La configuration des traceurs se fait via des fichiers texte se trouvant dans le système de fichiers debugfs. Aussi, la trace est produite dans le même système de fichiers. Les deux traceurs qui nous intéressent dans cette étude sont le traceur dynamique et le traceur d'événements.

Le traceur dynamique[32-34], comme son nom l'indique, permet l'insertion de points de trace dynamiques dans le noyau. Son implémentation se base sur Kprobes. La définition d'un point de trace se fait dans le fichier `/sys/kernel/debug/tracing/kprobe_events`. Il existe deux types de points de trace: les "probes" et les "return probes". Les "probes" peuvent être insérés à n'importe quel endroit dans le noyau. Il faut tenir compte des limites de kprobes bien-

sûr. Les “return probes” sont insérés aux points de sortie des fonctions et sont basés sur les kretprobes. Les points de trace dynamiques sont déclarés selon le format suivant:

p[:[GRP/]EVENT] SYMBOL[+offs]|MEMADDR [FETCHARGS] : pour les probes
r[:[GRP/]EVENT] SYMBOL[+0] [FETCHARGS] : pour les return probes

GRP permet d'assigner le point de trace à un groupe. EVENT correspond au nom du point de trace. SYMBOL et offs permettent de spécifier son adresse. Sinon MEMADDR permet aussi de spécifier l'adresse directement. FETCHARGS est la liste d'arguments à collecter. Les arguments suivants peuvent être collectés :

- %REG : un registre
- @ADDR : une adresse mémoire
- @SYM[+|-offs] : la mémoire se trouvant à l'adresse du symbole
- $stackN : la Nème entrée sur la pile
- $stack : l'adresse de la pile
- $retval : la valeur de retour d'une fonction

Une fois qu'un point de trace est défini, Ftrace crée un dossier portant le nom du probe dans le dossier `/sys/kernel/debug/tracing/kprobes`. Ce dossier contient quatre fichiers :

1. enabled : permet d'activer et de désactiver le point de trace. Ceci se fait en écrivant 1 ou 0 dans ce fichier. Il faut noter qu'il n'existe pas de moyen pour activer un groupe de points de trace en même temps. Il faut obligatoirement passer par le ficher enable de chaque point de trace.
2. format : décrit le format des données extraites à partir du point de trace.
3. filter : permet de définir les règles de filtrage pour cet événement. Les filtres servent de conditions pour les points de trace. Le principe des filtres est différent de celui des conditions utilisées dans les autres outils de traçage. En effet, ils ne peuvent pas employer les variables du code en utilisant les symboles et ne peuvent employer que les opérateurs logiques. Aussi, ils sont utilisés après que la collecte des données soit faite. Ainsi, les filtres ne permettent pas d'éviter de collecter des données inutilement, dans le cas où une

condition est fausse.

4. id : permet d'afficher l'identificateur du point de trace.

Ftrace est incapable de lire les informations de débogage du noyau. On ne peut pas donc définir un point de trace en spécifiant un numéro de ligne dans un fichier du code. On ne peut pas non plus collecter une variable en spécifiant son nom. On doit absolument déterminer son emplacement soit dans la mémoire soit dans les registres. Il faut donc passer par un outil de débogage. Aussi, Ftrace est incapable d'évaluer et de tracer des expressions employant des variables. Toutes les valeurs enregistrées sont représentées en hexadécimal dans la trace.

Le traceur dynamique de Ftrace offre alors peu de fonctionnalités par rapport à d'autres outils de traçage tels que SystemTap. Il nécessite une connaissance approfondie du code du noyau et l'utilisation d'un outil de débogage pour identifier les emplacements des variables et l'adresse correspondant à une ligne de code.

Le traceur d'événements[35] de Ftrace permet de se connecter aux points de trace statiques du noyau. Comme pour le traceur dynamique, la configuration et l'affichage de la trace se fait avec des fichiers texte. Pour chaque événement, il existe quatre fichiers semblables à ceux utilisés pour les points de trace dynamiques. Ftrace permet toutefois d'activer et de désactiver un groupe de points de trace statiques à la fois. Les points de trace sont listés dans le fichier available_events. Au moment de la compilation, Ftrace alloue pour chaque point de trace statique un espace dans une zone de mémoire statique. Cet espace permet de sauvegarder les détails du point de trace. Cet espace sera consulté par après pour pouvoir lister les points de trace.

Comme pour les points de trace dynamiques, Ftrace utilise les filtres à la place des conditions. Ces filtres ne permettent pas l'utilisation de n'importe quelle variable accessible au niveau du point de trace. Ceci est dû au fait que Ftrace n'a pas d'accès aux valeurs des registres au moment où le point de trace est rencontré et qu'il est incapable de lire les informations de débogage du noyau. Seules les variables transmises comme paramètre à la fonction enregistrée au point de trace peuvent être employées dans les filtres. En plus, seuls les opérateurs logiques et de comparaison peuvent être utilisés. Les opérateurs arithmétiques et de manipulation de bits ne sont pas supportés. En outre, les filtres sont appliqués après que les données soient collectées. Les filtres ne permettent pas alors d'éviter cette opération dans le cas où la condition est fausse. Le

temps pris par le handler de Ftrace dans le cas où un filtre est défini est quasiment le même indépendamment du résultat du filtre.

Le traceur d'évènements de Ftrace permet seulement la collecte des données telles que définies dans le TRACE_EVENT. Aucune autre donnée supplémentaire ne peut être collectée. Les données sont représentées dans la trace sous le format défini dans le TRACE_EVENT avec la macro TP_printk.

Ftrace définit pour chaque point de trace une fonction à lui seul, qui sera enregistrée auprès de l'événement si on veut l'activer. La fonction prend en paramètre les variables passées au point de trace et en extrait les données, tel que défini dans le TRACE_EVENT. La définition de cette fonction et des structures utilisées pour extraire et sauvegarder les données se fait en quatre étapes. Chaque étape consiste à la redéfinition de la macro TRACE_EVENT et des sous-macros qu'elle utilise. À la fin de chaque étape de redéfinition, le fichier d'en-tête contenant la définition du point de trace est réinclus. Ces étapes sont définies dans le fichier include/trace/ftrace.h. Ce fichier est lui-même inclus dans le fichier include/trace/define_trace.h.

À la fin de ces quatre étapes, nous aurons pour chaque point de trace la structure représentée par la macro TP_STRUCT__entry. Cette structure a les mêmes champs que ceux définis par la macro. Aussi, nous aurons la fonction qui accepte les paramètres du point de trace, remplit la structure en utilisant ces paramètres et enregistre cette structure dans les tampons de Ftrace. Ftrace se contente de sauvegarder la structure en binaire. Pour afficher la trace, Ftrace se base sur une autre structure produite au cours des quatre étapes pour garder les détails sur chaque champ de la première structure. Le format défini par la macro TP_printk est alors utilisé pour écrire la trace en format texte.

La figure suivante présente un exemple de trace générée par Ftrace en activant tous les points de trace statiques du système « sched » qui regroupe les points de trace définis au niveau de l'ordonnanceur. Chaque ligne de la trace correspond à un point de trace rencontré. Une ligne est composée du timestamp et du nom de l'évènement qui s'est produit. Le reste de la ligne correspond aux données collectées qui ont été définies avec la macro TP_printk.

```
<idle>-0 [003] 14282.836613: sched_stat_sleep: comm=compiz pid=1507
delay=20042550 [ns]
<idle>-0 [003] 14282.836617: sched_wakeup: comm=compiz pid=1507 prio=120
success=1 target_cpu=003
<idle>-0 [003] 14282.836628: sched_stat_wait: comm=compiz pid=1507 delay=0
[ns]
<idle>-0 [003] 14282.836630: sched_switch: prev_comm=kworker/0:1 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=compiz next_pid=1507 next_prio=120
compiz-1507 [003] 14282.836779: sched_stat_runtime: comm=compiz pid=1507
runtime=168425 [ns] vruntime=433338209014 [ns]
compiz-1507 [003] 14282.836790: sched_switch: prev_comm=compiz prev_pid=1507
prev_prio=120 prev_state=S ==> next_comm=kworker/0:1 next_pid=0 next_prio=120
<idle>-0 [000] 14282.836967: sched_stat_sleep: comm=Xorg pid=1047 delay=805854
[ns]
vlc-2883 [002] 14282.836968: sched_stat_runtime: comm=vlc pid=2883
runtime=856208 [ns] vruntime=415648939264 [ns]
<idle>-0 [000] 14282.836970: sched_wakeup: comm=Xorg pid=1047 prio=120
success=1 target_cpu=000
```

1.8 LTTng

LTTng(Linux Trace Toolkit Next Generation)[36-39] est un outil de traçage pour le système d'exploitation Linux. La version 2.0 de l'outil est capable d'insérer des points d'instrumentation dynamiques dans le noyau ainsi que de se connecter aux points de trace statiques définis avec TRACE_EVENT. La configuration de l'outil se fait en console en utilisant la commande « lttng » de l'outil « lttng-tools ».

Les points de trace dynamiques de LTTng sont implémentés en utilisant kprobes. Une seule fonction est enregistrée à tous les points de trace. À chaque fois qu'un point de trace est rencontré et que la fonction est appelée, celle-ci est capable de trouver l'évènement en question en se basant sur le paramètre de type « struct kprobe* » qui lui a été passé. En effet, ce paramètre est un membre d'une structure plus grande, interne à LTTng, qui garde tous les détails du point de trace.

À chaque fois que cette fonction est appelée, celle-ci ne fait qu'enregistrer l'adresse du point de trace dynamique qui a été rencontré. Aucune donnée supplémentaire n'est extraite. En effet, les points de trace dynamiques font partie des fonctionnalités récentes de LTTng et possèdent donc des fonctionnalités très limitées. Par conséquent, il n'est pas encore possible de collecter les valeurs des registres dans la trace et il n'est pas possible non plus d'évaluer des expressions définies avant le démarrage de la trace. Ceci est aussi dû au fait que LTTng n'est pas capable de lire les données de débogage du noyau comme SystemTap. Aussi, pour ces mêmes raisons,

LTTng ne permet pas non plus d'associer des conditions aux points de trace dynamiques ou même le filtrage de la trace comme le fait Ftrace.

Ainsi, la trace générée par LTTng ne contient que le timestamp collecté au moment de l'évènement, le nom de l'évènement ainsi que l'adresse de l'évènement. La figure suivante présente une trace collectée après avoir inséré trois événements dans différents endroits du noyau. L'outil « babeltrace » a été utilisé pour afficher la trace.

```
[447858717108] tp3: { 2 }, { ip = 0xFFFFFFFF81163D40 }
[447858802978] tp3: { 2 }, { ip = 0xFFFFFFFF81163D40 }
[447859173704] tp1: { 2 }, { ip = 0xFFFFFFFF8109C510 }
[447859540220] tp1: { 2 }, { ip = 0xFFFFFFFF8109C510 }
[447859669383] tp1: { 1 }, { ip = 0xFFFFFFFF8109C510 }
[447859671984] tp1: { 1 }, { ip = 0xFFFFFFFF8109C510 }
[447859690486] tp3: { 2 }, { ip = 0xFFFFFFFF81163D40 }
[447859707509] tp3: { 2 }, { ip = 0xFFFFFFFF81163D40 }
[447859715242] tp1: { 2 }, { ip = 0xFFFFFFFF8109C510 }
[447859750161] tp1: { 2 }, { ip = 0xFFFFFFFF8109C510 }
```

Les points de trace statiques de LTTng ne sont pas tout à fait ceux déclarés à l'aide de la macro TRACE_EVENT. En effet, LTTng n'utilise pas les fichiers d'en tête dans include/trace/events comme déclarations des points de trace. Il apporte quelques modifications à cette macro et aux autres macros utilisées pour déclarer les points de trace statiques afin d'optimiser leur intégration. En effet, comme nous l'avons noté précédemment, TRACE_EVENT est un mécanisme développé pour être adapté à Ftrace plus que les autres outils de tracage.

Ainsi, LTTng redéclare les points de trace dans d'autres fichiers d'en-tête propres à lui, en tenant compte de ces modifications. Par exemple, au niveau de la macro TP_fast_assign, au lieu de faire l'affectation à un membre de la structure __entry avec une affectation (e.g __entry->membre = variable;), LTTng utilise la macro tp_assign(e.g tp_assign(tid, t->pid)).

À partir de ces déclarations, LTTng utilise la même technique de réinclusion de fichiers d'en tête et de redéfinition des macros utilisée par Ftrace pour générer à l'aide du préprocesseur les fonctions à connecter à chaque point de trace ainsi que les structures intermédiaires utilisées pour extraire les données à partir des paramètres de la fonction. LTTng effectue cette opération en 10 étapes. Une fonction différente est ainsi définie pour chaque point de trace. La version actuelle de LTTng redéclare 51 points de trace statiques avec les nouvelles macros. Les autres points de trace ne sont pas encore accessibles.

Ces fonctions servent seulement à collecter les données du point de trace. Aucune autre donnée supplémentaire ne peut être collectée, telle que les registres ou une expression, et ceci pour les mêmes raisons qui limitent les fonctionnalités des points de trace dynamiques. En effet, LTTng n'apporte aucune modification au noyau Linux et ne peut donc pas avoir accès aux registres au moment où le point de trace est rencontré. De même, les points de trace statiques ne peuvent pas avoir de condition.

La figure suivante présente une trace produite en activant tous les 51 points de trace statiques redéfinis par LTTng. Chaque ligne représente un événement. Comme pour les points de trace statiques, chaque ligne commence par le timestamp de l'événement enregistré suivi du nom de l'évènement et de l'identificateur du processeur qui a rencontré le point de trace. Le reste de la ligne correspond aux données collectées au niveau du point de trace. On remarque que les données sont formatées exactement comme défini dans le TRACE_EVENT.

```
[3951188043258] sched_stat_wait: { 1 }, { comm = "kworker/1:0", tid = 2237,
delay = 0 }
[3951188044871] sched_switch: { 1 }, { prev_comm = "Xorg", prev_tid = 1047,
prev_prio = 20, prev_state = 0, next_comm = "kworker/1:0", next_tid = 2237,
next_prio = 20 }
[3951188045924] sched_stat_wait: { 2 }, { comm = "kworker/2:1", tid = 270,
delay = 0 }
[3951188046054] sched_stat_wait: { 0 }, { comm = "kworker/0:1", tid = 2443,
delay = 0 }
[3951188049056] sched_switch: { 2 }, { prev_comm = "kworker/0:1", prev_tid =
0, prev_prio = 20, prev_state = 0, next_comm = "kworker/2:1", next_tid = 270,
next prio = 20 }
[3951188049066] sched_switch: { 0 }, { prev_comm = "swapper", prev_tid = 0,
prev_prio = 20, prev_state = 0, next_comm = "kworker/0:1", next_tid = 2443,
next_prio = 20 }
[3951188050399] sched_stat_runtime: { 3 }, { comm = "kworker/3:0", tid = 15,
runtime = 29266, vruntime = 330737391980 }
[3951188055115] sched_stat_runtime: { 1 }, { comm = "kworker/1:0", tid = 2237,
runtime = 33153, vruntime = 338301744233 }
[3951188057701] sched_stat_wait: { 1 }, { comm = "Xorg", tid = 1047, delay =
33153 }
[3951188058963] sched_switch: { 1 }, { prev_comm = "kworker/1:0", prev_tid =
2237, prev_prio = 20, prev_state = 1, next_comm = "Xorg", next_tid = 1047,
next_prio = 20 }
[3951188064155] sched_switch: { 3 }, { prev_comm = "kworker/3:0", prev_tid =
15, prev_prio = 20, prev_state = 1, next_comm = "kworker/0:1", next_tid = 0,
next_prio = 20 }
[3951188065047] sched_stat_runtime: { 2 }, { comm = "kworker/2:1", tid = 270,
runtime = 32662, vruntime = 387855313303 }
[3951188067974] exit_syscall: { 1 }, { ret = 0 }
```

```
[3951188071953] sched_stat_runtime: { 0 }, { comm = "kworker/0:1", tid = 2443,
runtime = 39840, vruntime = 529594262442 }
```

LTTng sauvegarde la trace en binaire. Au cours des dix étapes de réinclusion/redéfinition, il génère la structure qui va servir à récupérer les données du point de trace à partir des paramètres de la fonction enregistrée. Aussi, une autre structure est générée qui décrit chaque membre de la première structure. Quand la trace est générée, la première structure est copiée en binaire dans les « ring buffers » de LTTng. Au moment de la lecture de la trace, la deuxième structure est utilisée pour lire correctement les données du ring buffer et pour générer la trace en format texte.

1.9 Valgrind

Valgrind[40] est un outil d'instrumentation binaire dynamique utilisé pour analyser des programmes en mode usager. Il s'agit d'un framework qui peut être utilisé pour bâtir toutes sortes d'outils d'analyse comme les profileurs. Il insère le code d'instrumentation dans le code original du programme quand celui-ci est en exécution. Aucune recompilation n'est nécessaire. Au cours de la session de traçage, seul Valgrind roule sur la machine. Le programme à instrumenter, lui, roule sur une machine virtuelle à l'intérieur de Valgrind. Toutes les instructions du processeur sont simulées. Valgrind peut utiliser soit les registres de la machine soit la mémoire pour simuler les registres du programme hôte.

Valgrind utilise une technique qui s'appelle D&R (Disassemble and resynthesize) : le code binaire est converti en un format intermédiaire auquel on ajoute du code d'instrumentation écrit dans ce même format. Le tout est alors reconverti en code binaire. Pour ce faire, un compilateur JIT (Just In Time) est utilisé. Le code intermédiaire est composé de blocs indépendants de l'architecture de la machine. Chaque bloc contient une liste d'instructions. Ces instructions peuvent contenir des expressions complexes.

L'insertion du code d'instrumentation se fait en huit étapes. D'abord, chaque instruction du code original est décomposée en un bloc d'instructions intermédiaires. Chaque instruction intermédiaire peut être représentée sous forme d'arbre. Ensuite, les arbres sont décomposés en sous-instructions représentées de façon séquentielle. À la fin de cette étape, le code original est transformé en instructions simples. À ce moment, ce code est passé à l'outil qui lui ajoute le code d'instrumentation. Le code obtenu est alors optimisé. Ensuite, des instructions sont regroupées

sous forme d'arbre pour éviter l'utilisation de variables temporaires. Ce code est alors converti en une liste d'instructions spécifiques à l'architecture en question. Les registres virtuels sont alloués. Finalement, les instructions sont converties en code binaire.

1.10 PIN

PIN[41] est un outil qui offre une API permettant la définition d'outils d'instrumentation dynamique de programmes en mode usager sous Linux. Un programme d'instrumentation écrit à l'aide de cet API est capable de contrôler l'exécution du programme à l'aide de ptrace. La définition du code d'instrumentation se fait à l'aide de fonctions définies dans ce programme. Ces fonctions peuvent être associées à une large liste d'évènements fournie par l'API de PIN.

L'instrumentation se fait à l'aide d'un compilateur JIT. Le compilateur prend en entrée des blocs du code binaire du programme à instrumenter et produit en sortie le code binaire instrumenté qui est presque identique à celui de l'entrée. L'instrumentation se fait à l'aide d'appels aux fonctions définies dans le programme d'instrumentation. Ces appels sont insérés dans le code binaire du programme instrumenté. L'état du programme est sauvegardé avant chaque appel à une fonction d'instrumentation, et est restauré au retour. Aucun format de représentation intermédiaire du code n'est utilisé. Les blocs de code de sortie sont sauvegardés dans une cache pour une réutilisation ultérieure.

1.11 Conclusion de la revue de littérature

Il existe donc une multitude d'outils de traçage dynamique et statique en mode noyau sur Linux. Toutefois, aucun d'entre eux ne semble posséder de flexibilité au niveau de l'utilisation d'expressions dynamiques au niveau des conditions et des actions exécutées tout en garantissant un temps d'exécution minimal.

Le module KGTP offre la flexibilité recherchée. Toutefois, il se limite au traçage dynamique. De plus, le temps d'exécution peut être amélioré. Dans ce qui suit, nous allons présenter une solution basée sur ce module mais qui offre aussi le traçage statique tout en garantissant des temps d'exécution largement meilleurs. Le chapitre suivant présente l'approche suivie et les résultats obtenus.

CHAPITRE 2 EFFICIENT CONDITIONAL TRACEPOINTS IN KERNEL SPACE

Abstract

With kernel static tracepoints defined using TRACE_EVENT, it is now possible to add instrumentation and obtain a low overhead trace of the whole system. However, these static tracepoints may be insufficient to diagnose the source of a functional or performance problem. Dynamic instrumentation fills the gap by making it possible to insert additional tracepoints in other locations at run time.

This article presents a new approach for tracing the Linux kernel with dynamic and static tracepoints. These tracepoints will be conditional. Conditions are defined using complex expressions that employ the code variables and make use of arithmetic and logic operations. These expressions are written using C-like syntax.

Both static and dynamic tracepoints will evaluate and collect expressions similar to those used for conditions. In addition, static tracepoints will collect the static tracepoint data as defined by TRACE_EVENT.

Our tool will be implemented based on GDB and KGTP, which is a GDB stub in kernel-space that partially implements dynamic tracepoints.

2.1 Introduction

With kernel static tracepoints defined using TRACE_EVENT and user-space tracepoints through the UST library, it is now possible to add instrumentation and obtain a low overhead trace of the whole system. However, these static tracepoinst may be insufficient to diagnose the source of a problem. Dynamic instrumentation fills the gap by making it possible to insert additional tracepoints in other locations at run time.

Recently, GDB was enhanced to support dynamic tracepoints in user-space. Using this feature, tracepoints can be defined in almost every location in a program. A set of actions can be associated with each tracepoint. These actions may be used to collect the values of the registers at

the time the tracepoint was hit or to evaluate user-defined expressions. These expressions may be complex and can employ all the program variables accessible from the tracepoint location. GDB being able to read the program debug information and to locate variables, we can refer to variables in these expressions by their names without having to care about their locations. GDB static and dynamic tracepoints may be conditional. In this case, expressions can be used as conditions. In order to simplify evaluation, GDB converts expressions used in conditions and actions to bytecode which is interpreted each time the corresponding tracepoint is hit. Moreover, in some situations, GDB converts the conditions' bytecodes into native code in order to improve performance.

More recently, the KGTP kernel module was submitted as a contribution to the Linux kernel. It uses kprobes to insert GDB dynamic tracepoints into the kernel, implements the RSP (Remote Serial Protocol) to communicate with GDB and can interpret the bytecode used by GDB to define conditions and actions, but is unable to convert this bytecode to native code.

The goal of this work is to extend the KGTP module by implementing a bytecode to native code converter in kernel space for both conditions and actions. GDB will also be integrated with TRACE_EVENT through KGTP in order to be able to list, enable and disable the kernel static tracepoints. Expressions may be used in conditions and additional actions. These expressions will be converted to native code and may employ each variable accessible from the static tracepoint location.

2.2 Previous Work

Dtrace is a tracing tool developed for the Solaris kernel and ported to other platforms such as Mac OS, QNX and FreeBSB. Static tracepoints can be inserted in the kernel source code using a C macro which expands to a non-existing function call. This function call is replaced by a NOP operation at link-time. The linker saves the function name and the call address. In order to enable the tracepoint, DTrace uses this saved information to replace the NOP by a probe which has the same name as the non-existing function.

Dynamic tracepoints are used to avoid the overhead caused by disabled static tracepoints. They can only be inserted in the kernel functions entry and exit points. They are implemented on the x86 architecture using a trap. When the tracepoint is hit, the instruction transfers control to DTrace.

Static and dynamic tracepoints in DTrace are defined using D scripts. D is a language which has a C-like syntax. Conditions can be associated to both tracepoints. D scripts are converted to an intermediate code. The intermediate language is a RISC instruction set. This code is emulated each time the tracepoint is hit.

SystemTap is a tracing tool similar to DTrace which implements dynamic and static tracepoints in the Linux kernel. SystemTap uses scripts in order to define both tracepoints. These scripts are compiled into kernel modules before tracing starts.

Dynamic tracepoints are implemented in SystemTap using kprobes. They can be conditional. Conditions are defined using C-like complex expressions. All the variables accessible from the dynamic tracepoint address can be used in these expressions. Dynamic tracepoints are able to evaluate and collect expressions that have the same characteristics as the expressions used for conditions.

SystemTap is also able to connect to the kernel static tracepoints defined using TRACE_EVENT. Conditional expressions are limited to using the variables passed to the SystemTap registered probe as defined in the TRACE_EVENT.

Ftrace is another Linux kernel tracing tool that is part of the mainline kernel. It offers both dynamic and static tracepoints. Dynamic tracepoints are based on kprobes. Because Ftrace is unable to read the kernel debug information, we have to manually specify the exact address location of each variable to collect.

Ftrace connects to the kernel static tracepoints defined with TRACE_EVENT. These tracepoints are only able to collect the data defined in the TRACE_EVENT declaration.

Ftrace is unable to associate conditions to both tracepoints. Only filters can be used. These filters have limited capabilities compared to SystamTap and GDB conditional expressions.

In the subsequent sections, we describe the architecture of the proposed solution. We explain the implementation of the bytecode to native code translator used in both dynamic and static tracepoints. We then describe the method used to integrate KGTP with the kernel static tracepoints and the modification we had to apply to the existing TRACE_EVENT implementation in order to collect the registers needed to evaluate expressions.

2.3 Methodology

2.3.1 Bytecode to native code translation

Converting the bytecode produced by GDB to native code has proved its efficiency in user-space especially for conditional tracepoints. In order to minimize the overhead of executing dynamic and static tracepoints probes in kernel space, we implemented the translator in KGTP.

Similarly to what GDBServer does in user-space, KGTP translates the bytecode using a one-to-one translation scheme, meaning that for each agent expressions opcode, there is a corresponding assembly code snippet[42].

Figure 5 shows the assembly code corresponding to the “add” opcode.

```
#define EMIT_ASM(BUFFER,INDEX,NAME, INSNS)
\
  do                                        \
    {                                       \
      extern unsigned char start_ ## NAME, end_ ## NAME;
\
      __asm__ ("jmp end_" #NAME "\n"                \
           "\t" "start_" #NAME ":"                  \
           "\t" INSNS "\n"                          \
           "\t" "end_" #NAME ":");                  \
    copy_into_buffer(BUFFER,INDEX,&start_ ## NAME,&end_ ##
NAME);\
    } while (0)

static void emit_add(unsigned char *dest, int *offset)
{
    EMIT_ASM (dest,offset,add,
            "add (%rsp),%r15\n\t"
            "lea 0x8(%rsp),%rsp\n\t"
            "dec %r14");
}
```

Figure 5 - "add" assembly code

As Figure 5 shows, each native code snippet is located in a separate function and is produced using inline assembly. The assembly code is located between two labels preceded by a jump to the second label. The jump instruction is useful to avoid executing that code when the function is called and the two labels are needed to get the start and end addresses of the native code in the program memory space. Having these two addresses, we just have to run what is in between to execute an “add” instruction.

In some situations, inline assembly is not sufficient to produce the correct native code. In fact, some opcodes have arguments which values change from an expression to another. Let's take the example of the “const8” opcode which pushes an eight-bit constant on the stack. This constant is provided in the bytecode and cannot be guessed when compiling the KGTP module. Thus, we have to overwrite the native code produced in order to put the right value. Figure 6 illustrates this case.

```
static void emit_const8(unsigned char *dest, int *offset,LONGEST arg)
{
    EMIT_ASM (dest,offset,const8_1,
            "movabs $0xffffffffffffffff,%r15");
    *((LONGEST*)(dest+*offset-8)) = arg;
}
```

Figure 6 - overwriting the native code

This is also the case for the “goto” and “if_goto” opcodes.

The native code translator that we implemented works in five steps:

- It allocates a virtually contiguous, executable memory. This buffer will contain the native code produced. It also allocates two other tables. The first one is a mapping table used to map the address of each opcode in the bytecode to the address of the corresponding native code snippet in the executable buffer. The second one is used to store the addresses of the “goto” and “if_goto” instructions in the bytecode. These two tables are used to determine and update the target addresses of the jump instructions used in the “goto” and “if_goto” opcodes.

- It starts by copying the function prologue into the executable buffer.
- It iterates over the bytecode and copies the corresponding native code snippets into the buffer. Overwrites are performed when needed (const8, zero_ext...).
- It copies the function epilogue into the buffer.
- It updates the jump addresses using the two tables.

Because the space allocated for the buffer is executable, the native code that it contains can be executed by casting the pointer to the buffer into a function pointer and calling that function.

Both dynamic and static tracepoints should be able to verify conditions and collect user defined expressions. Therefore, this translation is applied to the conditions and actions of all the tracepoints.

2.3.2 Listing and enabling static tracepoints

The user is not supposed to know the exact location and the name of every static tracepoint. We then have to be able to list the static tracepoints defined in the kernel upon request. Besides, because this information may change at any time and new static tracepoints can be added to the kernel, we cannot statically store it in the KGTP module. It must be retrieved at run time from the kernel.

Based on what is done in Ftrace, we used a static memory area in the kernel to store this information. By having the start and end addresses of this memory area, we can access it from both the kernel and the KGTP module, to read and write the information needed. We defined a structure that will contain all that data. For each static tracepoint, we have a corresponding instance of that structure that we called “kgtp_event_call” in the static memory area. Figure 7 shows that structure.

```
struct kgtp_event_call
{
    long int collect_regs;
    long int collect_sdata;
    void (*probe)(struct kgtp_event_call*, struct pt_regs,
char*);
    void *gentry;
    char *event_name;
    char *trace_system;
    void *address;
    int (*condition_function)(struct kgtp_event_call*,
struct pt_regs);
};
```

Figure 7 - structure used to store the tracepoint information

The collect_regs and collect_sdata members are used to determine whether or not we have to collect the registers and the static tracepoint data before executing the KGTP probe. The gentry pointer is used internally in KGTP and stores, among other things the pointers to the KGTP buffers. The event_name and trace_system contain the trace event name and system as defined in the TRACE_EVENT macro call. The address member contains the location of the static tracepoint in the kernel address space. Finally, the condition_function and probe function pointers contain the pointers to the condition function and KGTP probe respectively.

The structure data is written only at compile-time or by KGTP before starting the tracing session. Therefore, there is no data corruption risk in the case we have several threads accessing that data at the same time.

The structure is created at the static tracepoint call site. Figure 8 shows the code to create that structure. First of all, the tracepoint and the trace system names are stored in the corresponding sections. After that, we move to the _kgtp_event_calls section and create the kgtp_event_call structure. At that time, we can only write the tracepoint and trace system name and the tracepoint address which corresponds to the address from which the tracepoint condition is verified.

```
asm volatile(                                                           \
        ".ifndef __mstrtab_" __stringify(name) "\n\t"                   \
        ".section ___kgtp_event_calls_strings1,\"aw\",@progbits\n\t"    \
        "__mstrtab_" __stringify(name) ":\n\t"                          \
        ".string \"" __stringify(name) "\"\n\t"                         \
        ".previous\n\t"                                                 \
        ".endif\n\t"                                                    \
        );                                                              \
    asm volatile(                                                       \
        ".ifndef __mstrtab_" __stringify(TRACE_SYSTEM) "\n\t"           \
        ".section ___kgtp_event_calls_strings2,\"aw\",@progbits\n\t"    \
        "__mstrtab_" __stringify(TRACE_SYSTEM) ":\n\t"                  \
        ".string \"" __stringify(TRACE_SYSTEM) "\"\n\t"                 \
        ".previous\n\t"                                                 \
        ".endif\n\t"                                                    \
        );                                                              \
    asm volatile(                                                       \
        ".section _kgtp_event_calls,\"aw\",@progbits\n\t"               \
        "1:\n\t"                                                        \
        _ASM_PTR "0\n\t"                                                \
        _ASM_PTR "0\n\t"                                                \
        "19:\n\t"                                                       \
        _ASM_PTR "0\n\t"                                                \
        _ASM_PTR "0\n\t"                                                \
        _ASM_PTR "(__mstrtab_" __stringify(name) ")\n\t"                \
        _ASM_PTR "(__mstrtab_" __stringify(TRACE_SYSTEM) ")\n\t"        \
        _ASM_PTR "(18f)\n\t"                                            \
        _ASM_PTR "0\n\t"                                                \
        ".previous\n\t"                                                 \
```

Figure 8 - creating the kgtp_event_call structure

As the figure shows, the kgtp_event_call structures are created in a section called _kgtp_event_calls. Knowing the size of the structure and the start and end addresses of that section, we can iterate to list the static tracepoints. In order to do that, we had to modify the system image layout by adding the _kgtp_event_calls section to the linker script. We are then able to get the start and end addresses by calling the get_start_kgtp_event_calls and get_start_kgtp_event_calls functions. Figure 9 shows the modifications brought to the include/asm-generic/vmlinux.lds.h file.

```
#define KGTP_EVENT_CALLS()  VMLINUX_SYMBOL(__start_kgtp_event_calls) = .;    \
            *(_kgtp_event_calls)                \
            VMLINUX_SYMBOL(__stop_kgtp_event_calls) = .;

#define DATA_DATA                               \
    *(.data)                                \
    *(.ref.data)                            \
    *(.data..shared_aligned) /* percpu related */           \
    DEV_KEEP(init.data)                     \
    DEV_KEEP(exit.data)                     \
    CPU_KEEP(init.data)                     \
    CPU_KEEP(exit.data)                     \
    MEM_KEEP(init.data)                     \
    MEM_KEEP(exit.data)                     \
    . = ALIGN(32);                          \
    VMLINUX_SYMBOL(__start___tracepoints) = .;          \
    *(__tracepoints)                        \
    VMLINUX_SYMBOL(__stop___tracepoints) = .;           \
    /* implement dynamic printk debug */                \
    . = ALIGN(8);                           \
    VMLINUX_SYMBOL(__start___verbose) = .;                          \
    *(__verbose)                                                    \
    VMLINUX_SYMBOL(__stop___verbose) = .;               \
    LIKELY_PROFILE()                                \
    BRANCH_PROFILE()                        \
    TRACE_PRINTKS()                         \
                                    \
    STRUCT_ALIGN();                         \
    FTRACE_EVENTS()                         \
    KGTP_EVENT_CALLS();                     \
                                    \
    STRUCT_ALIGN();                         \
    TRACE_SYSCALLS()
```

Figure 9 - modifications brought to the linker script

Listing the static tracepoints is done in GDB using the “info static-tracepoint-markers”. Because GDB has no direct access to the tracepoint data in the static memory area, it communicates with KGTP using the appropriate requests to get that information. The two requests used in this case are “qTfSTM” and “qTsSTM”. The first request asks the remote stub, which is KGTP in our case, to return the information about the first static tracepoint. If the response that we return to GDB is successful, then GDB keeps sending “qTsSTM” requests and waiting for the response in order to get the information about the next static tracepoint. This process stops when we get to the end of the static memory area. In that case, we return an empty message to GDB.

2.3.3 Collecting the registers

In addition to the case when the user asks GDB to collect the registers using the “collect $regs” command, GDB may need the values of the registers at the moment the tracepoint was hit, in order to evaluate a condition or to evaluate an expression. GDB is able to find which register is used to store the value of a certain variable at that exact moment. Therefore, we have to collect the registers before calling the KGTP probe.

In order to avoid that the compiler insert any code which may modify the values of general purpose registers, the tracepoint address that is returned to GDB corresponds to the instruction that follows the code that does the collection. That way, we are always sure that the values recorded from the registers correspond to what GDB is asking for.

Registers collection is done using extended inline assembly. A pt_regs structure is provided as an input operand. That structure is created on the stack and not in the static memory area corresponding to that tracepoint, in order to avoid memory corruption issues in the case it is hit by multiple threads. Because we need at least one register to store the address of the pt_regs structure, we had to push the RAX register on the stack. After moving the reference to the pt_regs structure to that register, we pop the stored value directly to its corresponding location in the structure. We then copy the rest of the registers.

Collecting the registers is not always necessary. In fact, if the tracepoint is disabled or no tracepoint condition is specified, and only the tracepoint static data is collected, the registers

collected will not be useful. We thus check if the tracepoint is enabled and that KGTP needs the registers before collecting them. This is done using the “probe” and “collect_regs” fields in the kgtp_event_call structure. In the case the “probe” function pointer is NULL, this means that the KGTP probe is not registered and therefore, the tracepoint is disabled.

Because the assembly “TEST” instruction accepts only register operands, we used the same RAX register. After saving its value on the stack, we load the field in the register and call the test instruction. If saving the registers is needed, we then execute the code described in the previous paragraph. Otherwise, we directly call the KGTP probe without even restoring the RAX register. In fact, this register is listed as a clobbered register and therefore, the compiler will consider these changed. The same technique is used to test if the tracepoint is enabled.

2.3.4 Extracting the TRACE_EVENT data

Each TRACE_EVENT defines the parameters passed to the registered probes. The number and the types of these parameters vary from a tracepoint to another. Using a single probe for all the tracepoints is therefore unfeasible.

Therefore, each static tracepoint needs its own function that will accept the parameters and extract the appropriate fields. Writing such a function for each TRACE_EVENT manually can solve the problem but, in that case, KGTP won't be able to connect to the new static tracepoints added to the kernel, and we will find ourselves writing a new function each time we define a TRACE_EVENT. SystemTap suffers from this limitation. That function will extract the fields defined in the TRACE_EVENT from the parameters passed to it.

Based on the integration between TRACE_EVENT and Ftrace, we defined the functions to connect to the static tracepoints and the intermediate data we need using two stages. Figures 10 and 11 illustrate this.

```
#undef __array
#define __array(type, item, len)    type    item[len];

#undef __field
#define __field(type, item)     type    item;

#undef __field_ext
#define __field_ext(type, item, filter_type)    type    item;

#undef __dynamic_array
#define __dynamic_array(type, item, len) \
    type* item;\
    int item##_len;

#undef __string
#define __string(item, src) char* item;

#undef TP_STRUCT__entry
#define TP_STRUCT__entry(args...)   args

#undef TP_PROTO
#define TP_PROTO(args...)

#undef TP_ARGS
#define TP_ARGS(args...)

#undef TP_fast_assign
#define TP_fast_assign(args...)

#undef TP_printk
#define TP_printk(args...)

#undef TP_perf_assign
#define TP_perf_assign(args...)

#undef TRACE_EVENT
#define TRACE_EVENT(name, proto, args, tstruct, assign, print)          \
    struct kgtp_event_##name##__entry{                                  \
        tstruct                                                         \
    };


#undef DECLARE_EVENT_CLASS
#define DECLARE_EVENT_CLASS(name, proto, args, tstruct, assign, print)
\
    struct kgtp_event_##name##__entry{                                  \
        tstruct                                                         \
    };

#undef DEFINE_EVENT
#define DEFINE_EVENT(template, name, proto, args)

#include TRACE_INCLUDE(TRACE_INCLUDE_FILE)
```

Figure 10 - creating the __entry structure

```
#undef __dynamic_array
#define __dynamic_array(type, item, len)                                    \
    __entry->item##_len = len;

#undef __string
#define __string(item, src)

#undef __array
#define __array(type, item, len)

#undef __field
#define __field(type, item)

#undef __field_ext
#define __field_ext(type, item, filter_type)

#undef __assign_str
#define __assign_str(dst, src) __entry->dst = (char*)src;

#undef tp_assign
#define tp_assign(dest, src)        __entry->dest = src;

#undef tp_memcpy
#define tp_memcpy(dest, src, len)   memcpy(__entry->dest, src, len);

#undef tp_memcpy_dyn
#define tp_memcpy_dyn(dest, src, len) __entry->dest = src;

#undef tp_strcpy
#define tp_strcpy(dest, src)        __assign_str(dest, src);

#undef TP_fast_assign
#define TP_fast_assign(args...) args

#undef TP_PROTO
#define TP_PROTO(args...) args

#undef TP_ARGS
#define TP_ARGS(args...) args

#undef TRACE_EVENT
#define TRACE_EVENT(name, proto, args, tstruct, assign, print)          \
    void get_##name##_kgtp_string(char* buffer, proto){                 \
        struct kgtp_event_##name##__entry global__entry_##name;
\
        struct kgtp_event_##name##__entry *__entry =  &global__entry_##name;
\
        struct trace_seq __maybe_unused *p = &kgtp_seq_struct;          \
        tstruct;                                                            \
        assign;                                                         \
        snprintf(buffer,500,print);                                         \
    }
\
    EXPORT_SYMBOL(get_##name##_kgtp_string);
```

Figure 11 - function to extract the tracepoint data

In the first stage, we define the __entry structure as declared in the TRACE_EVENT. This structure contains all the fields that will be extracted from the parameters passed to the probe. As the figure shows, because we are only defining the structure here, we had to consider only the “name” and “tstruct” parameters of the TRACE_EVENT macro. The other tokens are simply ignored.

The first token is used as part of the structure name. The second one declares the fields inside the structure. By putting the tstruct token between the brackets({}), the preprocessor will define the structure fields using the corresponding macros(__array, __field, __dynamic_array...etc).

In the second stage, we create our function. The “name” token is pasted to the function name. The “proto” token is used to declare the function arguments list. The “tstruct” token is used to save dynamic arrays lengths if any. The “assign” token is used to generate the code to fill in the __entry structure. All the fields in the structure are a copy of the original parameters, except for dynamic arrays and strings. For these two cases, we wanted to avoid dynamically allocating memory, especially as these fields will be used only to generate the string and will no longer be needed afterwards. That is why we only copy the pointers to dynamic arrays and strings and use the original data without copying it. Finally, the “print” token is used to define the format string and to pass the appropriate arguments to the snprintf function that generates the function and copies it to a buffer.

Basically, the function creates the structure corresponding to that TRACE_EVENT, fills it using the parameters passed, generates the string using that structure and finally copies it to the buffer specified as a parameter to the function.

2.3.5 Condition evaluation and data collection

Recall that generating the string from the tracepoint parameters can only be done in the tracepoint site and cannot be moved to the KGTP module. In a previous implementation, we called the KGTP probe and passed the string we generated to it as a parameter. Because the condition was verified inside that probe, we found ourselves extracting the data for nothing in the case the condition was false. That is why we split the KGTP probe and implemented it in two functions.

The first function takes the pt_regs structure containing the saved registers and verifies if the condition is true. Then, depending on the result returned, we generate the TRACE_EVENT data and finally call the second KGTP function. Once again, in order to be thread safe, that data is passed to the function on the stack and not in the static memory area.

The second function executes the actions defined by the user one by one, similarly to what is done with dynamic tracepoints. In the case of the “collect $_sdata” action that collects the TRACE_EVENT data, KGTP copies the string collected on the tracepoint site to the buffers. First of all, KGTP needs to insert a frame head in that space to be able to identify that data when reading it afterwards. After that, it copies the string collected preceded by its length.

2.3.6 Algorithm description

In the previous sub-sections, we described the different parts of the algorithm used to collect TRACE_EVENTs data and to evaluate expressions in KGTP. In this sub-section, we will put these pieces together. The following pseudocode presents the modifications we had to apply to the static tracepoints sites in order to integrate KGTP with the Linux kernel:

```
create the kgtp_event_call structure

if (tracepoint_enabled)

        if (need_to_collect_the_registers)

                save the registers in the pt_regs structure

        if (no_condition OR condition_is_true)

                if (need_to_collect_TRACE_EVENT_data)

                        call the appropriate function to extract the data

                call the KGTP probe

call the old trace function
```

Other tracing tools like Ftrace are able to connect to static tracepoints using the functions provided by TRACE_EVENTs. Our situation was a bit more complicated. In fact, we needed to save the registers in order to be able to verify conditions and to evaluate expressions. These features are not implemented in Ftrace. We might think about registering the KGTP probe like the other tools and trying to recover the registers from the call stack, but we have to note that calling the function that extracts the data from the tracepoint parameters cannot be called from inside the probe, and thus we cannot avoid altering the code in the tracepoint site.

2.4 Results

The following section shows the results of running KGTP in both dynamic and static tracing modes. These results are compared with those obtained with SystemTap. Finally, we discuss the performance of combining KGTP and LTTng in static tracing mode with the performance of SystemTap.

In order to make sure the results are repeatable and accurate, we inserted dynamic and static tracepoints in a dummy function that does nothing but incrementing a counter. The function, called kgtp_test_function, was defined in kernel/module.c. The benchmark was executed with KGTP and GDB running on the same machine. This machine is equipped with an Intel Xeon E5405 (4 cores, 4 threads) at 2GHz and 8GB of memory.

The probes execution times were measured in cycles, with a kernel module that calls the test function 10.000 times. Figure 12 shows the test function and Figure 13 shows the loop used.

```
int kgtp_counter=1;
int kgtp_test_function(int counter1, int counter2)
{
    kgtp_counter++;
    __trace(kgtp_module_event,counter1,counter2);
}
```

Figure 12 - test function

```
set_current_state(TASK_INTERRUPTIBLE);

time1 = get_timestamp();
for (i = 0; i < NR_LOOPS; i++) {
    counter1++;
    kgtp_test_function(counter1, counter2);


    counter2++;
}
time2 = get_timestamp();
```

Figure 13 - loop used to calculate the execution times

2.4.1 Dynamic tracepoints with native code support

The goal of this test case is to compare the performance increase obtained by converting the bytecode produced by GDB into native code in dynamic tracing mode. The results are then compared to those produced by SystemTap where the probes are converted to compiled C code.

The test probe was inserted at the address of the instruction following the test function prologue for both KGTP and SystemTap. We first measured the execution time of the condition alone. In order to do that, we had to make sure the condition was always false. The second step was to measure the time needed to evaluate and store an expression. The last one was a combination of a true condition and an expression. The expressions used employ the two parameters of the dummy function (counter1 and counter2).

Figure 14 shows the GDB commands used to configure tracing and to output the trace for the third test case.

```
root@rafik-desktop:/usr/src/linux-2.6.37# gdb vmlinux
(gdb) target remote /proc/gtp
Remote debugging using /proc/gtp
0x0000000000000000 in ?? ()
(gdb) trace *0xffffffff810a3c50 if (2*counter1+3*counter2>0)
Tracepoint 1 at 0xffffffff810a3c50: file kernel/module.c, line 118.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect (2*counter1+3*counter2)
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind start
Found trace frame 0, tracepoint 1
#0 kgtp_test_function (counter1=1, counter2=0) at kernel/module.c:118
118   {
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
(2*counter11+3*counter2) = 2
(gdb) tfind
Found trace frame 1, tracepoint 2
118
(gdb) tdump
Data collected at tracepoint 1, trace frame 1:
(2*counter1+3*counter2) = 7
```

Figure 14 - GDB dynamic tracepoints

The two tools use Kprobes to connect dynamic tracepoints to the kernel. The tracepoint address was chosen so that Kprobes was able to optimize the probe and use a jump instead of the int3 interrupt. The execution times presented in the table below were calculated from the moment the

jump instruction was met until the execution of the original instruction. This was done by calculating the difference between the dummy function execution time with and without a kprobe connected. That means that the numbers presented below include the overhead added by kprobes.

Table 3 - execution times for dynamic tracepoints

	KGTP with native code (cycles)	SystemTap (cycles)
False condition : 2*param1+3*param2<0	202	351
Expression only : 2*param1+3*param2	500	1035
Condition and expression : 2*param1+3*param2>0	602	1061

The table shows that KGTP is always faster than SystemTap, whether to evaluate conditions or to store the expressions.

Based on these results, we may think that executing the native code produced by KGTP from the bytecode is faster than the optimized native code produced from the SystemTap script by the compiler. After analyzing the temporary C files produced by SystemTap, we discovered that the tool adds some “setup code” that is always executed at the start of the probe.

For KGTP, the execution time in the third test case is nearly the sum of the execution times of the first two test cases, if we eliminate the time taken to setup the kprobe(115 cycles for an optimized kprobe). That is not true in the case of SystemTap because of the setup code that is always executed.

Assuming that evaluating the conditions in the first and third cases takes nearly the same time, we can conclude that executing the native code produced by our implementation takes 102 cycles

for the two expressions. For SystemTap, the second and third cases let us conclude that the native code produced takes 26 cycles to be executed. This big difference may be explained by two reasons.

The first one is that the native code produced by KGTP is put in a function. Thus, we cannot neglect the execution time of the function prologue and epilogue. In the case of SystemTap, the condition evaluation and the data collection are done directly in the body of the kprobe handler.

The second one is that the native code produced by KGTP is a strict translation of the bytecode that GDB generated. Thus, its performance depends on how optimized the bytecode is. Let's take for example the condition used in the third test (2*param1+3*param2>0). GDB translates it to the following bytecode:

Table 4 - bytecode for the condition expression

Instructions	Description
0x22 0x02	const8 : push the 8-bit integer 2 on the stack, without sign extension
0x26 0x00 0x05	reg : push the value of the register 5, without sign extension
0x16 0x20	ext : pop an unsigned value from the stack. All bits to the left of bit 31(where the least significant bit is bit 0) are set to the value of bit 31.
0x04	mul : pop two integers from the stack, multiply them, and push the product on the stack.
0x16 0x20	ext
0x22 0x03	const8
0x26 0x00 0x04	reg
0x16 0x20	ext
0x04	mul
0x16 0x20	ext
0x02	add : pop two integers from the stack, and push their sum, as an integer.
0x16 0x20	ext
0x22 0x00	const8
0x2B	swap : exchange the top two items on the stack.
0x14	less_unsigned : pop two signed integers from the stack. If the next-to-top value is less than the top value, push the value one. Otherwise, push the value zero.
0x27	end : stop executing bytecode.The result should be the top element of the stack.

Let's take a look at the five “ext” instructions. All of them pop the value that was pushed in the previous instruction. Thus, we have five pop/push successive operations that we could avoid. In addition, the three “ext” instructions that follow the “const8” instruction are unnecessary because the constants that we pushed are already sign-extended to zero. We can then conclude that the bytecode produced by GDB could be optimized which can improve the native code significantly. Intermediate code optimization is not a new concept. It is used for example by Valgrind to make the intermediate representation of the original binary code more efficient.

Finally, we have to note the impact of the bad performance of KGTP buffers. In fact, the expression collected in the second test case is very similar to the condition evaluated in the first case. The main difference between their bytecodes is that the expression collected contains additional trace opcodes. These opcodes trace the values of the variables used in the expression. We notice that these opcodes are the main cause of the significant difference between these two cases. Thus, by using more efficient data structures, like ring buffers to store the trace, the time taken to execute the “trace” and “trace_quick” opcode can be reduced. Moreover, by integrating KGTP with other tracing tools like LTTng, we can make use of their fast tracing capabilities. Storing the trace would be then performed by LTTng.

2.4.2 Static tracepoints

The static tracepoint was created by defining a new TRACE_EVENT, as shown in figure 2. The expression defined in this static tracepoint is very similar to the one defined in the dynamic tracepoints test. It collects the two parameters passed to the dummy function.

KGTP is able to connect to the static tracepoint and collect the data as defined by the TP_printk macro automatically. On the other side, SystemTap is only able to trace the parameters given to the probe registered to the tracepoint automatically. In order to get the same results with the two tools, we have to redefine the way to extract the data in the SystemTap script. Figures 4 and 3 show the KGTP and SystemTap scripts used to connect to the tracepoint.

As a first step we tried to measure the overhead of a disabled static tracepoint in the kernel. After that, we compared the performances of KGTP and SystemTap.

```
TRACE_EVENT(kgtp_module_event,

    TP_PROTO(int counter1, int counter2),

    TP_ARGS(counter1, counter2),

    TP_STRUCT__entry(
        __field(    int,    counter1    )
        __field(    int,    counter2    )
    ),
```

Figure 15 - TRACE_EVENT used for the test

```
(gdb) target remote /proc/gtp
Remote debugging using /proc/gtp
0x000000000000000 in ?? ()
(gdb) info static-tracepoint-markers
Cnt          ID                              Enb    Address                          What
1            module::kgtp_module_event       n      0xffffffff810a3cef in kgtp_
test_function at kernel/module.c:120                       Data: ""
2            module::module_load             n      0xffffffff810a6b33 in load_
module at jernel/module.c:2691                             Data: ""
3            skb:net_dev_xmit                n      0xffffffff814be486 in dev_h
ard_start_xmit at net/core/dev.c:2069                      Data: ""
4            skb::net_dev_xmit               n      0xffffffff814be72c in dev_h
ard_start_xmit at net/core/dev.c:2046                      Data: ""
(gdb) strace *0xffffffff810a3cef if (2*counter1+3*counter2>0)
Probed static tracepoint marker "module::kgtp_module_event"
Static tracepoint marker 1 at 0xffffffff810a3cef: file kernel/module.c, line 120.
(gdb) actions
Enter action for tracepoint 1, one per line.
End with a line saying just "end".
>collect (2*counter1+3*counter2)
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind start
Found trace frame 0, tracepoint 1
#0 0xffffffff810a3cef in kgtp_test_function (counter1=1, counter2=0)
       at kernel/module.c:120
120                __trace(kgtp_module_event,counter1,counter2);
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
(2*counter1+3*counter2) = 2
(gdb) tfind
Found trace frame 1, tracepoint 1
0xffffffff810a3cef        120          __trace(kgtp_module_event,counter1,counter2);
(gdb) tdump
Data collected at tracepoint 1, trace frame1:
(2*counter1+3*counter2) = 7
```

Figure 16 - GDB static tracepoints

```
probe kernel.trace("kgtp_module_evnet")
{
    if(2*$counter1+3*$counter2>0)
        printf ("time=%d count=%d\n",$counter1,$counter2)
}
```

Figure 17 - SystemTap script

2.4.2.1 KGTP kernel overhead

In order to avoid collecting the registers when the KGTP tracepoint is disabled, we check if the KGTP probe was registered in the static memory area corresponding to the tracepoint that was hit. We wanted to measure the overhead caused by this additional code. In order to do that, we calculated the execution time of the dummy function before and after the changes were applied to the kernel. Table 2 shows the results.

Table 5 - KGTP kernel overhead

	Function execution time(cycles)
Before	12
After	19

We may conclude that a disabled static tracepoint costs an extra 7 cycles. This can be explained by the fact that we need at least one register to proceed to the check. This is why we are saving the RAX register before calling the TEST instruction.

Saving the RAX register was unavoidable. We may want to put it in the clobbered registers list to force the compiler to use other registers for the kernel variables, but in the case we asked GDB to collect the registers using the « collect $regs » command in the same static tracepoint, the traced value of RAX would be incorrect.

2.4.2.2 KGTP vs. SystemTap

As for dynamic tracepoints, we calculated the results for the three test cases. Table 3 shows the results we made. We have to note that our proposed extended KGTP with static tracepoints also supports our proposed bytecode to native code translator.

Table 6 - KGTP vs. SystemTap

	KGTP(cycles)	SystemTap(cycles)
condition	154	223
data	1216	1252
condition and data	1368	1336

The table shows that KGTP is faster than SystemTap in the cases where the condition is false or there is no condition. We may think that the execution time in the case of a probe which condition is true will nearly be the sum of the two first cases. This is only true for KGTP. Indeed, as for dynamic tracepoints, the SystemTap probe always executes the « setup code ». That explains the fact that SystemTap is faster than KGTP in the third case.

We also notice that the two tools are too slow to extract and save the data. In the case of KGTP, this is caused by the way the data is generated and also by the way it is stored in the buffers. Unlike more optimized tools like Ftrace or LTTng, KGTP fills the __entry structure and then pretty prints it to generate a string as defined in the TP_printk macro.

Once the string is generated, it is copied in the KGTP buffers after the appropriate space is allocated. It is clear that KGTP is not well suited for high performance tracing and that storing the binary __entry structure instead of the string and using more efficient data structures to record the trace will improve the performance of the tool.

We run a similar benchmark for Ftrace to show the difference of performance between storing strings in simple buffers and storing binary data in more efficient ring buffers. We used the same test module to run three test cases on our static tracepoint. In the first one, only the data was recorded. In the second and third ones, we associated a filter to the tracepoint. We could not use the same expression used with KGTP and SystemTap expressions, because arithmetic operators are not permitted by Ftrace. We used a simpler expression instead. Table 4 shows the results.

Table 7 - Ftrace execution time

	Execution time
Data only	297
False filter	360
True filter	370

The table shows that Ftrace is nearly four times faster than KGTP when collecting the trace data and storing it in the ring buffer. This proves that the current implementation of KGTP is not that optimal.

Moreover, static tracepoint conditions suffer from the same optimization problems that we discussed in the dynamic tracepoints section. By implementing the native code optimizer, we shall have faster executions times for expressions in static tracing mode.

Finally, we have to note that KGTP has another advantage compared to all the other tracing tools. In fact, thanks to the changes we applied to the kernel in order to collect the registers at the tracepoint site, and because GDB is able to read the debug information generated at compile-time, static tracepoint conditions may use all the global and local variables accessible from the tracepoint address. Moreover, in addition to the static tracepoint data defined in the TRACE_EVENT call, GDB static tracepoints are able to execute other actions like collecting the registers and evaluating user defined expressions, as for dynamic tracepoints. Other tracing tools have limited capabilities compared to KGTP. SystemTap is limited to using the parameters passed to the registered function in the condition and in the expressions to collect. Ftrace and LTTng do not even implement conditions and are only able to collect the static tracepoint string.

2.4.2.3 KGTP-LTTng integration

Currently, the time taken to have the static tracepoint data written in the KGTP buffers from the moment the tracepoint is hit can be calculated using the following equation:

$T = T_{reg} + T_{cond} + T_{gen} + T_{buf}$

T_{reg} is the time needed to verify whether the static tracepoint is enabled and to collect the registers if needed.

T_{cond} is the time taken to check if a condition is associated to the tracepoint and to evaluate it.

T_{gen} is the execution time of the get_##name##_kgtp_string function. Finally, T_{buf} is the time needed to store the string in the KGTP buffers.

Knowing that the lack of performance of our implementation is primarily caused by the way we are generating and storing data, we thought about combining the flexibility of GDB agent expressions and the high performance of LTTng. Instead of generating strings, LTTng is able to store the __entry structure into its ring buffers directly. Similarly to the way we defined the function that generates the string from the TRACE_EVENT using macro redefinitions, LTTng is able to store the __entry structure metadata for every static tracepoint. That way, it is able to extract the appropriate data from the structure and produce the pretty printed string when the user is reading the trace.

For each static tracepoint, LTTng defines a function that is used to extract and record the trace static data. These functions can replace the get_##name##_kgtp_string probes used in the current implementation and also the KGTP probe that stores the data in case we want to collect only the tracepoint static data. The algorithm used at the tracepoint site becomes:

```
create the kgtp_event_call structure

if (tracepoint_enabled)

    if (need_to_collect_the_registers)

        save the registers in the pt_regs structure

    if (no_condition OR condition_is_true)

        call the old trace function
```

In that case, KGTP is in charge of generating the tracepoint condition native code and evaluating it using the registers collected. By registering the LTTng function to the tracepoint, it will be called by the old trace function. Based on the results presented in table 3, and assuming that evaluating the true condition (2*counter1+3*counter2>0) takes the same time as a false condition (2*counter1+3*counter2<0), we can conclude that the time needed to evaluate the registers and to evaluate the condition at the tracepoint site is equal to the execution time presented in table 3 in the case we have a false condition, which is 154 cycles. We can then extrapolate these results to measure the new execution time. The equation above becomes:

$T = T_{reg} + T_{cond} + T_{lttng}$

$T_{reg}+ T_{cond}$ is the time needed to collect the registers and to evaluate the condition by KGTP and T_{lttng} is the time taken by LTTng to collect the tracepoint data. T_{lttng} was measured on a vanilla kernel where we defined the same static tracepoint used for the other test cases. The same test module was used to make the measurements. The table below presents the extrapolated results:

Table 8 - KGTP-LTTng integration

	KGTP+LTTng(cycles)	SystemTap(cycles)
condition	154	223
data	333	1252
condition and data	487	1336

With this implementation, the call to the original trace function is performed only if the KGTP condition is true. Therefore, this mechanism can be used by all the tracing tools that can register to TRACE_EVENT and is not limited to LTTng.

2.5 Conclusion

We have described an implementation based on the existing KGTP kernel module and GDB that offers conditional dynamic and static tracepoints. Conditions are defined using complex C-like expressions that can use all the variables accessible from the tracepoint address. All the

arithmetic and logic operations are supported by KGTP. Both dynamic and static tracepoints are able to evaluate and save the values of user-defined expressions similar to those used in the conditions.

Additionally, the tool is able to collect static tracepoints data as defined by the TRACE_EVENT macro. Unlike SystemTap, our implementation is able to extract the data manually without the need to redefine that data. With the ability of inserting dynamic tracepoints or reusing static tracepoints, and to specify arbitrary conditions and data collection expressions, our initial objectives have been achieved.

Even though we showed that our implementation is faster than SystemTap for dynamic tracepoints and has comparable execution times for static tracepoints, we suffered from the low performance of KGTP buffering scheme and some optimizations are required in order to reach the performance of other tools such as Ftrace and LTTng.

As we showed earlier, the bytecode produced by GDB for the expressions used in the conditions and actions can be optimized by eliminating unneeded operations.

Static data extraction can also be optimized. Instead of generating and copying strings into the trace buffers, we can simply save the intermediate structure used to extract the data and use it to generate the string only when the user is viewing the trace.

Finally, the results also show that the data structures used to store the trace are not optimized and can be replaced by more efficient structures like ring buffers.

CHAPITRE 3 DISCUSSION GENERALE

Dans cette section, nous allons revenir sur les résultats présentés dans l'article. D'autres éléments de discussion vont aussi être traités. Finalement, une comparaison fonctionnelle avec les autres outils de traçage est étudiée.

3.1 Performance

Nous avons conclu dans l'article que KGTP présente une solution de traçage plus rapide que SystemTap pour les points de trace dynamiques et de niveau comparable à celui-ci pour les points de trace statiques. Cependant, ces résultats restent nettement insatisfaisants comparés aux performances d'autres outils de traçage tels que Ftrace et LTTng. Il est donc impératif de penser à améliorer les performances de KGTP si on veut en faire un outil de choix.

Pour accélérer l'exécution du code natif, il faut implémenter un optimiseur de code intermédiaire. Tel que nous avons discuté dans l'article, il existe des opérations qui ne sont pas toujours nécessaires tels que l'empilement et le dépilement successifs d'une seule variable. Ces opérations sont exécutées à la fin de chaque opération arithmétique et logique. Nous épargnerons ainsi un temps d'exécution important en éliminant ces opérations.

Pour les points de trace statiques, la sauvegarde des données extraites par TRACE_EVENT sous forme de chaine de caractères s'est avérée inefficace par rapport au stockage sous forme binaire. Vu que la chaine est obtenue en se basant sur la structure binaire, on pourrait copier celle-ci directement dans les tampons de KGTP. Cette méthode ne sera certainement pas triviale à implémenter. En effet, chaque évènement ayant sa propre structure, il va falloir identifier le type de la structure se trouvant dans le tampon au moment de l'affichage de la trace. Pour ce faire, nous allons devoir garder les détails de chaque membre de la structure dans une autre structure qui sera utilisée par la suite. La génération de la chaine de caractères se fera alors au besoin au moment de l'affichage de la trace. Cette technique est utilisée par LTTng et Ftrace.

Finalement, les structures de données utilisées comme tampons de trace se sont avérées inefficaces. Il faudra penser à les optimiser. D'autres outils tels que LTTng utilisent des structures plus performantes, des tampons circulaires sans verrou.

3.2 Fonctionnalités

La trace produite par KGTP ne peut être lue que ligne par ligne et ce, à l'aide de commandes de GDB. Cette méthode peut être suffisante dans le cas où la trace est de petite taille. Cependant, elle est insuffisante pour les traces de grandes tailles. Il faudra donc trouver une autre méthode pour afficher la trace. Les autres outils de traçage utilisent par exemple la console pour l'affichage en temps réel comme SystemTap, sans avoir à taper de commandes, ou encore l'affichage dans un fichier texte comme dans Ftrace.

L'enregistrement de la trace dans un fichier est une fonctionnalité manquante dans GDB. Seul le traçage en mémoire est offert. Ceci limite l'utilisation de l'outil pour des fins de débogage et les données ne peuvent pas être exploitées facilement pour des analyses plus approfondies du système comme c'est le cas pour d'autres outils comme LTTng.

Finalement, les données enregistrées dans la trace ne donnent aucune indication sur le temps. On peut penser à inclure le timestamp dans la trace. On peut aussi inclure l'identificateur du processeur ayant atteint le point de trace.

3.3 Intégration avec les outils de traçage

Nous avons proposé une méthode d'intégration de KGTP avec LTTng. Cette méthode permet de se connecter aux points de trace statiques. Toutefois, elle permet seulement de collecter les données définies avec TRACE_EVENT. Les probes de LTTng sont incapables de collecter des données supplémentaires ce qui n'est pas le cas de KGTP.

De plus, il est clair que cette structure n'est pas facile à utiliser puisqu'il faut configurer le traçage avec GDB et KGTP, puis effectuer le traçage avec LTTng. Dans ce cas, il ne sera pas possible de lire la trace à partir de GDB, puisqu'il ne peut pas communiquer directement avec LTTng.

Il serait plus intéressant de copier les fonctionnalités utilisées dans LTTng. Nous aurons à implémenter la lecture des informations de débogage, effectuer les modifications au niveau des points de trace statiques pour collecter les registres. Des commandes supplémentaires nous

permettraient de définir des actions supplémentaires pour les points de trace statiques et dynamiques.

CONCLUSION

La complexité accrue des systèmes informatiques rend l'identification des causes de disfonctionnement et des problèmes de performance de plus en plus difficile. Il est clair que les outils classiques qui étaient efficaces pour des systèmes séquentiels ne sont plus d'une grande utilité pour les systèmes parallèles, distribués ou temps-réel. C'est pour cette raison que les outils de traçage et de débogage doivent suivre le rythme de cette évolution.

Nous avons présenté dans ce mémoire un travail basé sur GDB et KGTP qui propose une implémentation des points de trace statiques et dynamiques conditionnels. Cette implémentation ne requiert pas d'outils additionnels tels qu'un compilateur pour son fonctionnement. Les conditions associées aux points de trace peuvent être complexes, employer la plupart des opérateurs logiques et arithmétiques et surtout faire référence à toutes les variables accessibles à partir de l'adresse du point de trace. Les fonctions enregistrées au niveau des deux types de points de trace ont accès à la valeur de tous les registres du processeur au moment où le point de trace a été atteint. Elles sont capables alors d'évaluer et de sauvegarder dans la trace toutes les expressions définies par l'usager.

Bien que SystemTap offre ces options pour les points de trace dynamiques, nous avons démontré que le travail proposé présente des temps d'exécution meilleurs. Pour les points de trace statiques, notre solution est la seule actuellement à pouvoir associer des conditions qui peuvent utiliser toutes les variables du code et ne se limite pas aux variables passées à la probe comme c'est le cas pour SystemTap. Aucun autre traceur n'implémente les points de trace conditionnels pour le noyau Linux. Nous pouvons donc conclure que nous avons pu atteindre les objectifs de ce projet au niveau fonctionnel.

L'importance de la haute performance des outils de traçage est incontestable. Il est important que la charge causée par le traçage lui-même n'ait pas d'impact sur le système à tracer. Évidemment, KGTP demeure inefficace quand il s'agit de traçage de haute performance. Il devient alors impératif de repenser l'implémentation de KGTP sur laquelle nous nous sommes basés afin de l'optimiser et éliminer ces problèmes de performance. Une meilleure implémentation des fonctions de collecte des données définies dans TRACE_EVENT est aussi importante.

LTTng étant plus développé et adapté au traçage que GDB, nous pensons que l'intégration de notre travail dans LTTng lui-même, en plus de l'implémentation de la lecture des informations de débogage, serait la solution la plus efficace. Ainsi, nous allons pouvoir profiter de ces fonctionnalités au niveau des outils d'analyse qui sont développés pour LTTng tels que la détection d'intrusion et l'analyse de dépendances.

BIBLIOGRAPHIE

[1] (2012/01/16). *Tracing book*. Available: http://lttng.org/tracingwiki/index.php/TracingBook

[2] R. A. Robert W. Wisniewski, Mathieu Desnoyers, Maged M.Michael, Jose Moreira, Doron Shiloach, and Livio Soares, "Experiences understanding performance in a commercial scale-out environment," in *International Euro-Par Conferenc*, 2007.

[3] D. Toupin, "Using Tracing to Diagnose or Monitor Systems " *Software, IEEE* vol. 28, pp. 87-91, 2011.

[4] (2012/01/16). *UST Official Website*. Available: http://lttng.org/ust

[5] J. Corbet. (2008, 2012/01/16). *Tracing: no shortage of options*. Available: http://lwn.net/Articles/291091/

[6] (2011, 2012/01/16). *KGTP patch*. Available: http://lwn.net/Articles/430666/

[7] S. Goswami, "An introduction to KProbes," 2005.

[8] (2012/01/16). *Kprobes Documentation in the Kernel*. Available: http://lxr.free-electrons.com/source/Documentation/kprobes.txt

[9] P. P. A. Mavinakayanahalli, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes," in *Ottawa Linux Symposium*, 2006.

[10] (2012/01/16). *Debug Exceptions*. Available: http://www.logix.cz/michal/doc/i386/chp12-03.htm

[11] M. Hiramatsu. (2010, 2012/01/16). *The Enhancement of Kernel Probing-Kprobes Jump Optimization*. Available: http://lttng.org/tracingwiki/images/f/fa/HiramatsuLinuxCon2010.pdf

[12] J. Corbet. (2007, 2012/01/16). *Kernel markers*. Available: http://lwn.net/Articles/245671/

[13] J. Corbet. (2009, 2012/01/16). *Fun with tracepoints*. Available: http://lwn.net/Articles/346470/

[14] J. Corbet, "On the value of static tracepoints," 2009.

[15] S. Rostedt. (2010, 2012/01/16). *Using the TRACE_EVENT() macro*. Available: http://lwn.net/Articles/379903/

[16] R. P. R. Stallman, et S. Shebs, *Debugging with GDB: the GNU Source-Level Debugger for GDB*: Free Software Foundation, 2010.

[17] R. P. R. Stallman, et S. Shebs. (2012/01/16). *GDB Remote Serial Protocol*. Available: http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html#Remote-Protocol

[18] R. P. R. Stallman, et S. Shebs. (2012/01/16). *GDB fast tracepoints*. Available: http://sourceware.org/gdb/onlinedocs/gdb/Create-and-Delete-Tracepoints.html

[19] R. P. R. Stallman, et S. Shebs. (2012/01/16). *General Bytecode Design*. Available: http://sourceware.org/gdb/current/onlinedocs/gdb/General-Bytecode-Design.html#General-Bytecode-Design

[20] R. P. R. Stallman, et S. Shebs. (2012/01/16). *Bytecode descriptions*. Available: http://sourceware.org/gdb/current/onlinedocs/gdb/Bytecode-Descriptions.html#Bytecode-Descriptions

[21] (2012/01/16). *Dtrace page in the Tracing Wiki*. Available: http://lttng.org/tracingwiki/index.php/DTrace

[22] M. W. S. a. A. H. L. Bryan M. Cantrill, "Dynamic Instrumentation of Production Systems," in *USENIX Annual Technical Conference*, Boston, MA, USA, 2004.

[23] (2012/01/16). *The GNU Awk User's Guide*. Available: http://www.gnu.org/software/gawk/manual/gawk.html

[24] W. Waite. (2012/01/16). *ANSI C Specification*. Available: http://eli-project.sourceforge.net/c_html/c.html

[25] F. C. Eigler, "Problem Solving With Systemtap," presented at the Red Hat Summit 2007, San Diego, 2007.

[26] (2012/01/16). *Systemtap Official Website*. Available: http://sourceware.org/systemtap/

[27] W. C. V. Prasad, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Ottawa Linux Symposium*, 2005.

[28] J. Corbet. (2007, 2012/01/16). *On DTrace envy*. Available: http://lwn.net/Articles/244536/

[29] V. P. Frank Ch. Eigler, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, Brad Chen, "Architecture of systemtap: a Linux trace/probe tool," 2005.

[30] M. Wielaard. (2009, 2012/01/16). *A SystemTap update*. Available: http://lwn.net/Articles/315022/

[31] (2012/01/16). *Ftrace Documentation in the Kernel*. Available: http://lxr.free-electrons.com/source/Documentation/trace/ftrace.txt

[32] J. Corbet. (2009, 2012/01/16). *Dynamic probes with ftrace*. Available: http://lwn.net/Articles/343766/

[33] (2012/01/16). *Ftrace Dynamic Tracepoints Documentation*. Available: http://lxr.free-electrons.com/source/Documentation/trace/kprobetrace.txt

[34] M. Hiramatsu. (2010, 2012/01/16). *Dynamic Event Tracing in Linux Kernel*. Available: https://events.linuxfoundation.org/slides/lfcs2010_hiramatsu.pdf

[35] S. Rostedt. (2010, 2012/01/16). *Ftrace: Now and Then*. Available: http://lttng.org/tracingwiki/images/e/e6/RostedtLinuxCon2010.pdf

[36] M. Desnoyers, "Low-Impact Operating System Tracing," Doctorat, École Polytechnique de Montréal, 2009.

[37] (2012/01/16). *LTTng Official Website*. Available: http://lttng.org/lttng2.0

[38] M. D. Martin Bligh, Rebecca Schultz, "Linux Kernel Debugging on Google-sized clusters," presented at the Ottawa Linux Symposium, 2007.

[39] M. R. D. Mathieu Desnoyers, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," presented at the Ottawa Linux Symposium, Ottawa, Canada, 2006.

[40] J. S. Nicholas Nethercote, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, California, USA, 2007.

[41] R. C. Chi-Keung Luk, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi and Kim Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN conference on Programming language design and implementation*

[42] I. Corporation. (2012/01/16). *Intel 64 and IA-32 Architectures Software Developer Manuals* Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html